

GPES: A GENERAL PROCESS ENGINEERING SYSTEM

by

MOHAMMAD SHARIF ARAB-ISMAILI

B.S., Abadan Institute of Technology
(1971)


SUBMITTED IN PARTIAL FULFILLMENT
OF THE REQUIREMENTS FOR THE
DEGREE OF

DOCTOR OF PHILOSOPHY

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

May 1978

Signature of Author: 
Department of Chemical Engineering, May 12, 1978

Certified by: 
J.H. Porter, Thesis Supervisor

Accepted by: 
G.C. Williams, Chairman, Department Committee on Graduate Theses

GPES: A GENERAL PROCESS ENGINEERING SYSTEM

by

Mohammad Sharif Arab-Ismaili

Submitted to the Department of Chemical Engineering
on May 12, 1978 in partial fulfillment of the requirements
for the Degree of Doctor of Philosophy

ABSTRACT

The General Process Engineering System (GPES) has been designed as a generalized framework which can be used to create different kinds of simulation systems for process engineering, each suited to the needs of a particular set of users. This has been accomplished by having generalized data structures represent the elements of a chemical process flowsheet such as units, streams, components, etc. However, to create a particular simulation system, one must define these data structures and provide computer programs to perform process unit operations. The information set defining a process element is called a template. Hence a system created by GPES is called a Template Based System (TBS). A language called Template Definition Language (TDL) has been provided to enable the administrator of the TBS to communicate with the system to define templates.

Once a TBS has been implemented (templates have been defined and computational subroutines provided) a process designer may use that system to model, simulate, or design any arbitrary process configuration. A language called Process Engineering Language (PEL) has been provided to enable the designer to communicate with the TBS in terms very similar to those he might use to describe the process to another designer. The system provides the user with some other advanced features not usually found in other general purpose process simulators. It provides an environment wherein the designer and the computer can work as partners on the problem solving team, each performing the job he does best, enhancing the capability of either partner working alone.

The created systems (TBS's) are not limited to a particular class of processes. They are open-ended systems which can be easily modified, expanded, or updated.

This thesis describes the motivation and design of GPES and demonstrates its application by developing a number of prototype TBS's. The system has been implemented in PL/1 on Multics, a time-sharing system by Honeywell.

Thesis Supervisor: James H. Porter
Lecturer, Department of Chemical Engineering

Department of Chemical Engineering
Massachusetts Institute of Technology
Cambridge, Massachusetts 02139
May 12, 1978

Professor George C. Newton, Jr.
Secretary of the Faculty
Massachusetts Institute of Technology
Cambridge, Massachusetts 02139

Dear Professor Newton:

In accordance with the regulations of the Faculty, I hereby submit a thesis entitled "GPES: A General process Engineering System," in partial fulfillment of the requirements for the Degree of Doctor of Philosophy in Chemical Engineering at the Massachusetts Institute of Technology.

Respectfully submitted,

Mohammad Sharif Arab-Ismaili

ACKNOWLEDGMENTS

I wish to express my sincerest gratitude to my thesis supervisor, Professor James H. Porter. His valuable guidance and timely suggestions deserve appreciation of the highest kind. I am also grateful to Professor L.B. Evans of the MIT Chemical Engineering Department, to Professor S.E. Madnick of the MIT Sloan School of Management, and to Professor W.D. Seider of the University of Pennsylvania, for their help while serving on the thesis committee.

Acknowledgment is also due to Professor J.F. Louis, and Dr. J. D. Teare of the MIT Energy Laboratory for their valuable suggestions and comments concerning this work.

I am grateful to Dr. H. Cohen and Mr. B. Misra for their effort in testing the system by implementing a prototype TBS for MHD process studies.

I wish to thank Dollina Borella, Lana Krasner, Patricia Rynne, Alice Sanderson, and Barbara Thomas for their effort and patience in typing this manuscript.

Finally, I wish to express my indebtedness to my wife and my mother who have patiently and lovingly encouraged me throughout my studies.

This work was supported by the MIT Energy Laboratory and the U.S. Department of Energy.

Mohammad Sharif Arab-Ismaili
Cambridge, Massachusetts
May 1978

TABLE OF CONTENTS

	<u>Page</u>
1. SUMMARY	22
1.1 Introduction	22
1.2 Criteria for Computer Systems Amenable to Simulation and Design	24
1.3 Existing Computer Systems and the Problem	27
1.4 Thesis Objective	28
1.5 Thesis Work	28
1.5.1 General Process Engineering System	28
1.5.2 Development Process of a Template Based System	34
1.5.3 Development of a Number of Prototype TBS's	37
1.5.4 An Example: The Development of a Prototype TBS for Analyzing Heat Exchanger Networks	38
1.6 Conclusions and Thesis Contributions	54
2. INTRODUCTION	59
2.1 Chemical Process Simulation	59
2.2 Simulation Versus Design	59
2.3 Criteria for Computer Systems Amenable to Simulation and Design	61
2.4 The Problem	64
2.5 Thesis Objective	68
2.6 Thesis Work	68
2.7 General Process Engineering System	69
2.8 Operating Environment	73
2.9 Organization of the Thesis	74
3. A FRAMEWORK FOR THE DEVELOPMENT OF GENERAL PURPOSE PROCESS SIMULATORS	76
3.1 A Chemical Process	76

	<u>Page</u>
3.1.1 Chemical Components	77
3.1.2 Streams	79
3.1.3 Units	81
3.2 Other Elements of Interest	83
3.2.1 Functions	84
3.2.2 Variables	85
3.3 Units of Measurement	85
3.4 Calculating Routines	86
3.4.1 Route Selection	88
3.5 Template Based System	90
4. THE TEMPLATE DATA BASE	92
4.1 Information Content of a Template Data Base	92
4.1.1 The Dimension Table	99
4.1.2 Stream Templates	102
4.1.3 Unit Templates	106
4.1.4 Component Templates	110
4.1.5 Function Templates	113
4.1.6 Table of Property Estimation Methods	117
4.1.7 Control Information	117
4.1.8 Text File	125
4.2 The Template Data Base Segments	127
4.3 The Template Definition Language	131
4.4 "update_tdb Program	135
4.4.1 Consistency of the Data in the Template Data Base	137
4.5 Protection of the Template Data Base	138
4.5.1 Utility Programs	139
4.5.1.1 gaccess_tdb Program	139

	<u>Page</u>
4.5.1.2 taccess_tdb Program	141
4.5.1.3 copy_tdb Program	142
4.5.1.4 copy_seg Program	143
4.5.1.5 delete_tdb Program	
5. DATA STRUCTURES REPRESENTING A PROCESS FLOWSHEET	144
5.1 Memory Management	144
5.1.1 The Process Directory Data Structure	147
5.2 Data Structures Representing the Process Elements	149
5.2.1 Parameters Structure	149
5.2.2 The Unit Structure	152
5.2.3 The Component Structure	155
5.2.4 The Stream Structure	160
5.2.5 The Pre-Defined Function Structure	162
5.2.6 The User-Defined Function Structure	162
5.2.7 The Variable Structure	169
5.2.8 The Data Structure Containing the Property Estimation Methods in Use	169
5.3 Process Files	174
5.4 Component Files	177
6. TBS PROGRAMS	184
6.1 Primary Programs	184
6.1.1 Calculating Routines	184
6.1.2 Pre-Defined Function Evaluation Routines	187
6.2 Secondary Programs	187
6.3 Interaction between TBS Programs and the GPES Executive	187
6.4 Writing a TBS Program	197

	<u>Page</u>
6.4.1 Input	197
6.4.2 Output	197
6.4.3 Error Detection	198
6.4.4 Convergence Routines	199
6.4.5 Writing an "all" Calculating Routine	201
6.5 Service Routines	202
7. PROCESS ENGINEERING LANGUAGE -- BASIC PRINCIPLES	212
7.1 Basic Concepts of PEL	212
7.1.1 Command Objects	212
7.1.2 Process Files	218
7.1.3 Component Files	218
7.1.4 Property Estimation Methods	218
7.1.5 Units of Measurement	220
7.1.6 Profile Parameters	220
7.1.7 Arithmetic Expressions	220
7.1.8 Command Elements	221
7.2 Classification of Commands	222
7.2.1 Configuration Commands	222
7.2.2 Value Assignment Commands	223
7.2.3 Output Commands	224
7.2.4 Input Commands	224
7.2.5 Clearing and Switching Commands	225
7.2.6 Commands for Component Files	225
7.2.7 Commands for Process Files	226
7.2.8 Continue Command	226
7.2.9 Iterative Commands	226

	<u>Page</u>
7.3 Using a TBS	228
7.4 Using the System	228
7.5 PEL Messages	230
7.5.1 Information Messages	231
7.5.2 Requesting Messages	231
7.5.3 Warning Messages	231
7.5.3.1 Informatory Warning Messages	233
7.5.3.2 Severe Warning Messages	233
7.5.4 Error Messages	233
7.5.4.1 Informatory Error Messages	233
7.5.4.2 Severe Error Messages	236
7.5.4.3 Calculate Error Messages	236
7.5.5 System Messages	236
7.5.5.1 Informatory System Messages	237
7.5.5.2 Severe System Messages	237
8. GPES ADMINISTRATION AND PROTECTION	238
8.1 The GPES Text File	238
8.2 The TBS Table	239
8.3 The Users Table	244
8.4 Various Copies of GPES Files	249
8.5 The GPES Organization	252
9. THE GPES EXECUTIVE	256
9.1 The Program Structure	260
9.2 Lexical Analysis Phase	264
9.3 Command Recognition Phase	268
9.4 Syntax Analysis Phase	270

	<u>Page</u>
9.4.1 Intermediate Form of Arithmetic Expressions	275
9.4.2 Intermediate Forms of Commands	292
9.5 Execution Phase	307
10. EXERCISE IN THE DEVELOPMENT OF TEMPLATE-BASED SYSTEMS	308
10.1 Development Process of a Template-Based System	309
10.2 Development of a Prototype TBS for Analyzing Heat Exchanger Networks	312
10.3 A Prototype TBS for Hydrocarbon Processes	330
10.4 A Prototype TBS for Magnetohydrodynamic Processes	359
11. RECOMMENDATIONS AND CONCLUSIONS	364
11.1 Recommendations	364
11.2 Conclusions and Thesis Contributions	371
APPENDIX A: STATE OF THE ART	376
A.1 Structure of the Programs	381
A.2 Input Methods	383
A.3 Data Checking	383
A.4 The Storage of Data	383
A.5 Operating Modes	384
A.6 Available Unit Models	384
A.7 Physical Property Determination	384
A.8 Convergence Acceleration	384
APPENDIX B: <u>TEMPLATE DEFINITION LANGUAGE</u>	386
Reserved Words in TDL	386
Program interrupt	389

	<u>Page</u>
TDL Commands	389
delete Commands	390
end Command	390
insert Commands	391
list Commands	397
print Commands	397
replace Commands	399
revise Commands	400
APPENDIX C: TBS SERVICE ROUTINES	404
C.1 Basic Service Routines	405
C.2 Comparison Service Routines	413
C.3 Service Routines Retrieving or Storing the Values of Parameters	416
C.4 Service Routines Retrieving Other Variables of Interest	442
C.5 Service Routines Interacting with the User	445
C.6 Service Routines Performing Arithmetic Operations on Two Parameter Sets	448
APPENDIX D: PROCESS ENGINEERING LANGUAGE -- DETAILED DESCRIPTION	460
D.1 COMMAND ELEMENTS	460
Character Set	460
Symbols	462
Literals	462
Terminal Symbols	462
Identifiers	463
Language Keywords	465
Established Identifiers	467
User Supplied Identifiers	475
Composite Identifiers	476
Blanks	477
Comments	477
D.2 EXPRESSIONS	478
Use of Expressions	478
Expression Operations	478
Arithmetic Operations	479

	<u>Page</u>
Boolean Operations	479
Comparison Operations	480
Combination of Operations	480
Priority of Operators	482
Operands of an Expression	483
D.3 FUNCTIONS	484
Pre-defined Functions	484
User-Defined Functions	485
Built-In Functions	486
D.4 SEMI-FORMAL DEFINITION OF <u>PEL</u> SYNTAX	492
D.5 <u>PEL</u> COMMANDS	509
assume commands	509
bugs command	509
calculate commands	510
clear command	514
close command	514
connect command	515
continue command	517
copy commands	517
create commands	518
delete commands	521
deletef commands	524
disconnect command	525
end command	526
escape command	526
help command	526
include command	528
leave command	528
let commands	529
leta commands	533
list commands	533
listf commands	535
listt commands	536
load commands	536
loop command	538
news command	539
open commands	539
print commands	541
printf commands	547
printt commands	548
profile command	549
read commands	553
reada commands	558
repeat command	558
save commands	565
specify commands	566
stop command	570
terminate command	571
unspecify commands	572
use command	576

	<u>Page</u>
D.6 <u>PEL</u> MESSAGES	578
D.6.1 Warning Messages	579
D.6.1.1 Informatory Warning Messages	579
D.6.1.2 Sever Warning Messages	588
D.6.2 Error Messages	590
D.6.2.1 Informatory Error Messages	590
D.6.2.2 Severe Error Messages	599
D.6.2.3 Calculate Error Messages	616
D.6.3 System Messages	620
D.6.3.1 Informatory System Messages	620
D.6.3.2 Severe System Messages	623
 APPENDIX E: LITERATURE CITATIONS	 631
 BIOGRAPHICAL SKETCH	 644

LIST OF TABLES

<u>Table</u>	<u>Page</u>
4.1 The Value Status Codes Table	100
4.2 The Template Data Base Segments	130
4.3 The Template Definition Language Commands	132
6.1 Number of Primary Programs in a TBS	189
6.2 Call Statements Used by GPES Executive to Invoke a TBS Primary Program	191
6.3 External Variables for Use by TBS Programs	195
6.4 Service Routines Retrieving or Storing the Values of Parameters	205
6.5 A Short Description of Each Service Routine	207
7.1 PEL Commands	213
7.2 Built-In Constants	219
7.3 Format of PEL Messages	232
7.4 Conditions in an Expression Producing Severe Warning Messages	234
7.5 States Where Informatory Error Messages May Be Produced	235
8.1 Commands Used by the GPES Administrator to Update the TBS Table	243
8.2 Commands Used by the GPES Administrator to Update the Users Table	248
8.3 The GPES Files	251
8.4 GPES Programs Used by Each Group	255
9.1 Token Class Types	266
9.2 Command Object Codes	271
9.3 Command Verb Codes and Object Flags	272

<u>Table</u>	<u>Page</u>
9.4 Operations in the Parsed Matrix	278
9.5 The Representation of Operands in the Parsed Matrix	280
9.6 The Index of Built-In Functions in the Parsed Matrix	281
9.7 The Representation of Operators in the Parsed Matrix	282
9.8 The Precedence Table	285
9.9 The Control Blocks of Various Commands	294
10.1 TBS-II Property Estimation Methods	344
A.1 Computer-Aided Process Design Systems	378
B.1 The TDL Commands	387
B.2 Permitted Abbreviations in the TDL Commands	388
D.1 Special Characters	461
D.2 Functions of Other Special Characters	464
D.3 Language Keywords	468
D.4 General Format of PEL Commands	498

LIST OF FIGURES

<u>Figure</u>	<u>Page</u>
1.1 The Hierarchical Structure of GPES Usage	30
1.2 Use of the GPES	32
1.3 The Development Process of a TBS	35
1.4 Heat Exchangers TBS -- Countercurrent Exchanger Model	39
1.5 Heat Exchangers TBS -- Mixer Model	40
1.6 Heat Exchangers TBS -- Divider Model	41
1.7 Heat Exchangers TBS -- Unit Convergence Model	43
1.8 Heat Exchangers TBS -- Printout of Some Templates	45
1.9 Heat Exchangers TBS -- Heatex Calculating Routine	47
1.10 Heat Exchangers TBS -- The Process Flowsheet for the Example Case	49
1.11 Heat Exchangers TBS -- A PEL Computer Session for the Example Case	50
1.12 Heat Exchangers TBS -- Another PEL Computer Session for the Example Case	51
1.13 Heat Exchangers TBS -- Adjuster Model	53
3.1 An Example of Routing Problem	89
4.1 The Parameter Template Structure	94
4.2 The Parameter Value Status Structure	98
4.3 The Dimension Table Structure	101
4.4 An Example of a Dimension Table	103
4.5 The Stream Template Structures	104
4.6 An Example of a Stream Template	107
4.7 The Unit Template Structures	108
4.8 An Example of a Unit Template	111

<u>Figure</u>	<u>Page</u>
4.9 The Component Template Structures	112
4.10 An Example of a Component Template	114
4.11 The Function Template Structures	115
4.12 An Example of a Function Template	118
4.13 The Property Estimation Methods Table Structure	119
4.14 An Example of a Property Estimation Methods Table	120
4.15 The Control Information Structure	121
4.16 An Example of a Control Information	126
4.17 Format of the Text File	128
4.18 An Example of a Text File	129
4.19 The Flow Chart for update_tdb Program	136
4.20 Information Flow Regarding the Template Data Base	140
5.1 The Process Directory Data Structure	148
5.2 An Example of a Process Directory Data Structure	150
5.3 The Parameters Structure	151
5.4 The Unit Structure	153
5.5 An Example of a Unit Structure	156
5.6 The Component Directory Structure	157
5.7 An Example of a Component Directory	159
5.8 The Stream Structure	161
5.9a An Example of a Stream Structure with No Components	163
5.9b An Example of a Stream Structure with Some Components	164
5.10 The Pre-Defined Function Structure	165
5.11 An Example of a Pre-Defined Structure	166
5.12 The User-Defined Function Structure	167
5.13 An Example of a User-Defined Function Structure	170

<u>Figure</u>	<u>Page</u>
5.14 The Variable Structure	171
5.15 An Example of a Variable Structure	172
5.16 The Data Structure for Property Estimation Methods in Use	173
5.17 An Example of the Data Structure for Property Estimation Methods in Use	175
5.18 The Directory of a Process File	176
5.19 An Example of a Process File	178
5.20 The Component File Structure	180
5.21 The Structure of a Component in the Component File	181
5.22 An Example of a Component File	183
6.1 The Interaction between the GPES Executive and TBS Programs	185
6.2 Classification of TBS Programs	188
6.3 The Arguments Structure	193
6.4 Interaction between GPES Executive, a Unit Calculating Routine, and Service Routines	194
6.5 Classification of Service Routines Retrieving or Storing the Values of Parameters	203
8.1 Format of the GPES Text File	240
8.2 The TBS Table	241
8.3 TBS Registration Form	245
8.4 The Users Table	246
8.5 GPES User Registration Form	250
8.6 Information Flow Regarding GPES Files	253
8.7 The Hierarchical Structure of GPES Usage	254
9.1 The Flow Chart for System Initialization Phase	257

<u>Figure</u>	<u>Page</u>
9.2 The Flow Chart for TBS Attachment Phase	258
9.3 The Flow Chart for Process Initialization Phase	259
9.4 The Layout of the "pel" Program	262
9.5 The Token Table	265
9.6 An Example to Demonstrate the Function of Lexical Analysis Phase	269
9.7 The Flow Chart for Command Recognition Phase	274
9.8 The Parsed Matrix Structure	276
9.9 An Example of a Parsed Matrix	283
9.10 The Stack Used in Parsing the Expressions	286
9.11 The Parser Algorithm	287
9.12 The Command Header Structure	293
9.13 The Cblock Structure	295
9.14 The Cblock1 Structure	297
9.15 The Cblock2 Structure	298
9.16 The Cblock3 Structure	299
9.17 The Cblock4 Structure	301
9.18 The Cblock5 Structure	303
9.19 The Cblock6 Structure	304
9.20 The Cblock7 Structure	305
10.1 Development Process of a Template-Based System	310
10.2 Heat Exchangers TBS -- Countercurrent Exchanger Model	314
10.3 Heat Exchangers TBS -- Mixer Model	315
10.4 Heat Exchangers TBS -- Divider Model	316
10.5 Heat Exchangers TBS -- Unit Convergence Model	318
10.6 Heat Exchangers TBS -- Printout of Some Templates	320

<u>Figure</u>	<u>Page</u>
10.7 Heat Exchangers TBS -- Heatex Calculating Routine	322
10.8 Heat Exchangers TBS -- Process Flowsheet for the Example Case	324
10.9 Heat Exchangers TBS -- A PEL Computer Session for the Example Case	325
10.10 Heat Exchangers TBS -- Another PEL Computer Session for the Example Case	326
10.11 Heat Exchangers TBS -- Adjuster Model	328
10.12 Heat Exchangers TBS -- Inserting a Template	329
10.13 TBS-II -- Templates for Control Information, Dimension Table, and Property Estimation Methods Table	331
10.14 TBS-II -- Stream Template	333
10.15 TBS-II -- Component and Function Templates	335
10.16 TBS-II -- Isothermal Flash Unit Template	337
10.17 TBS-II -- Calculating Routines for Stream and Isothermal Flash Unit	338
10.18 TBS-II -- Distillation Unit Template	341
10.19 TBS-II -- Function Calculating and Evaluating Routines	348
10.20a TBS-II -- Process Flowsheet for the Example Case	349
10.20b TBS-II -- Computational Flowsheet for the Example Case	349
10.21 TBS-II -- A PEL Computer Session for the Example Case	351
10.22 TBS-II -- Another PEL Computer Session for the Example Case	355
10.23 MHD TBS -- Process Flowsheet of an Open Cycle MHD Topped Plant	360
10.24 MHD TBS -- A PEL Computer Session for the Example Case	362
10.25 MHD TBS -- Another PEL Computer Session for the Example Case	363
C.1 Examples of the Use of Service Routines Related to Unit Data Structures	407
C.2 Examples of the Use of Service Routines Related to Component Data Structures	408

<u>Figure</u>	<u>Page</u>
C.3 Examples of the Use of Service Routines Related to Function Data Structures	411
C.4 Examples of the Use of Service Routines Related to Stream Data Structures	412
C.5 Examples of the Use of Service Routines Related to Flow Parameters	414
C.6 Examples of the Use of Service Routines Directly Retrieving or Storing the Values of Parameters	418
C.7 Example of the Use of the Service Routine Retrieving the Arguments	444
C.8 Example of the Use of the Service Routine Retrieving the Property Estimation Methods in Use	444
C.9 Examples of the Use of Service Routines Performing Arithmetic Operations on Two Parameter Sets	450
C.10 Examples of the Use of Service Routines Performing Arithmetic Operations on Flow Parameters of Two Streams	455

CHAPTER 1SUMMARY1.1 Introduction

Since the advent of large scale computers workers in almost every professional discipline have attempted, in some degree, to formalize their rules and analysis procedures to enhance the effectiveness of computers within their discipline. For many professions, a large class of problems exist which we shall call network analysis problems. These networks are characterized by streams which transport items obeying the general laws of conservation (i.e. Rate of Input - Rate of Output + Rate of Production - Rate of Depletion = Rate of Accumulation) and by process units, which collect these items from in-flowing streams and redistribute the items to out-flowing streams. The rules that govern the proportion and rate of distribution of each item type are usually defined as the Science of that discipline.

Thus, in electrical engineering for instance, coulombs are conserved and capacitors, resistors, conductor elements, etc. are process units in electrical networks. Raw materials, finished goods, dollars are conserved items and factories are process units in economic networks. Mass of chemical species, energy and momentum are conserved items and distillation columns, heat exchangers, pumps, etc. are process units in chemical process networks. It is possible to continue almost indefinitely in this manner, cataloging networks of single disciplines or combined networks of multidisciplines.

Two questions are normally asked concerning networks. The first question is, given a specified network, how will it respond under given

perturbations in stream item flows or in process unit parameters which govern the distribution proportion or rate? This is normally called the simulation problem. It is characterized as the analysis of fixed network modules in a fixed configuration. Much time has been devoted to developing procedures or methodology for solving problems of this type. The results of these efforts have led to rules to govern simulation procedures. These rules have been incorporated into computer systems which we shall classify as General Purpose Process Simulators. Many such simulators are in existence and they have characteristically required tens of man-years to implement.

The second question normally asked concerning networks is, given a specified response of items in out-flowing streams from the network and given specified constraints on the parameters of process units or streams within the network, which network will behave in the desired manner? This is called the design problem. The design problem may have many solutions but as the number of constraints on the system behavior is increased, the number of solutions is diminished until there may be no solutions. These problems are characterized by the fact that in arriving at a solution not only the process units must be selected, but their configuration within the overall process must also be chosen. Methodology for selecting process units and units arrangement is not yet formalized although efforts have been devoted to developing General Purpose Process Synthesizers since the early 1970's. At present computer aided design systems must include man in the loop to carry out the important task of process unit selection and arrangement to accomplish a stated objective. It has not yet been established that these "creative" aspects of design, much of which are developed through experience (design know-how), can be formalized to the

extent that they can be incorporated in a computer system which would eliminate man from the loop. Serious doubts are entertained as to whether this accomplishment is at all possible without subjecting the computer to the impossible task of analyzing the doubly infinite set of considering all possible process unit selections and configurations. Thus, the current procedure to solve design problems is for man to propose the process unit selection and arrangement, use a General Purpose Simulator to analyze the proposed network, and then compare the proposed network's response to the desired response and alter either network configuration or process unit selection, or both, until a network is developed with the desired characteristics. The process of design is necessarily iterative, and having arrived at a solution one is usually not certain that other solutions do not exist.

Considering the iterative nature of design problems, a general purpose simulator to be effective as a design tool must use a mode of operation, methods of input and calculation techniques that minimize the effort required for designer-computer communication, so as to maximize effective interaction between the designer and the computer. The time scale is important in process design.

These characteristics not only enable the simulator to provide the design atmosphere for process designers, but they also enhance the capability and effectiveness of the system as a tool for simulation.

1.2 Criteria for Computer Systems Amenable to Simulation and Design

There are two basic modes of computer operation: the off-line batch processing mode and the on-line interactive processing mode.

On-line processing enables the designer to speed up the design process. However, replacing the batch mode by the interactive mode is not

sufficient. The interface between the human mind and the computer must also be modified to make the job easier for the designer, as discussed by Porter [133]. The essence of his discussion is that "it is necessary to create an atmosphere wherein the design team and the computer as an added partner are able to rapidly evaluate the effects of equipment arrangement and process variable selection in chemical process design".

The basic question is, "How does the designer communicate with and maintain control over the computer?" This is the area where most automated design systems are inefficient. The designer is required to prepare some sort of input forms specifying the process configuration, the operating conditions of the units and the thermodynamic states of the streams. After the input is fed, the computer takes over complete control and arrives at a solution. Such a system is not very conducive to creative design. The environment of process design is much more dynamic. A designer rarely knows the entire plant configuration at the outset of a design. The flowsheet is more likely to evolve from an initial concept to a final design after various perturbations of the initial concept. He must have more contact with and greater control over the computer. He must be allowed to specify what the computer should do at each stage of the design and receive feedback in terms of the intermediate results so that he can decide upon the next action to be taken by the computer.

One method of achieving this interaction is to provide a process design language which is more "natural" to the designer and can be interpreted by the computer. The language must be such that he can describe flowsheet structure, request unit calculations, provide input, request output and have the computer carry out other instructions. Thus, he may solve a full flowsheet or any of its parts that he desires. He

should also be able to change the flowsheet structure easily without respecifying the whole flowsheet.

The language should permit choice of engineering units for input data to speed up the man-machine communication. The language should accept arithmetic expressions where numerical data are expected. The user should have access to every piece of information about the process network and be able to refer to them in arithmetic expressions. In this way the user can relate different items of information about the process flowsheet. The system should allow the repetitive execution of a group of commands to enhance the designer's iterative search. The system should allow the user to save the results of one analysis in a file to be used as input for further analysis. This will save both the designer and the computer time in not having to reinitiate the problem.

To perform process engineering effectively, the engineering data necessary for design must be available. The computer can assist engineers in analyzing, estimating, and retrieving these data. However, it is not efficient to have many individual computer-aided systems for simulation, design, physical property estimation, analyzing laboratory data and so on. Therefore, such systems should be organized into an integrated system with a common data base, so that the consistency of these data is maintained throughout the various stages of process engineering.

The description of a flowsheet should be independent of the programs for analysis. The same description should serve all analysis for which that flowsheet is applicable. For example, it should be suitable for steady state simulation, equipment sizing, and economic evaluation, as noted by Evans and Sieder[33] who discuss the requirements of an advanced computer system which will be needed to solve the process engineering

problems of the 1980's. Such a system must be extendable and capable of modification. It should be easy to add new types of process units, to define new types of streams, species, etc. Hence, the system can be expanded to analyze new types of processes. The inability of existing systems to analyze new energy conversion processes such as the magnetohydrodynamics (MHD) process has been due to the lack of this feature in these systems.

1.3 The Problem

Despite the considerable effort expended by the chemical and petroleum companies on development of computer-aided design programs, the process engineering profession still lacks the computing tools needed by process designers. The existing systems don't meet the criteria discussed earlier. Hence, they don't provide the design atmosphere for process engineers. On the other hand, existing systems are only applicable to those process flowsheets having conventional vapor-liquid streams. This inflexibility makes it either impossible or very difficult to expand present-day systems to include other types of processes such as coal processing systems.

The difficulties may be traced to the following characteristics of existing programs:

- a. They are not interactive.
- b. They are not integrated.
- c. They do not provide a natural problem oriented language.
- d. They do not have the capability of dynamic modification of the process network.
- e. They do not have the capability of storing the output model for future study.
- f. They are mostly developed for simulation, not for design.
- g. They are not flexible.

1.4 Thesis Objective

The objective of the thesis was to develop a framework for the development of general purpose process simulators that:

- a) are applicable to all types of chemical processes,
- b) are more adaptable to the design environment.

1.5 Thesis Work

A general framework for modeling of chemical processes has been found and based upon this framework, a computer system called General Process Engineering System (GPES) has been designed and implemented. GPES allows any group or organization to create its own computer aided design system for engineering of chemical processes. These systems will not have the shortcomings of existing systems that were discussed earlier. These systems could be very simple or very sophisticated depending on the particular needs and applications of the group or organization. Such systems meet the criteria discussed earlier and therefore, provide the design atmosphere for process designers. These systems are not limited to analyzing process flowsheets having only conventional vapor-liquid streams. They are applicable for analyzing any type of process flowsheet including energy conversion processes such as coal gasification and MHD. Such systems could easily be modified, expanded, and updated. This thesis describes the design and application of GPES. Using GPES several prototype computer aided design systems have been created to demonstrate the application and use of GPES and systems created by GPES.

1.5.1 General Process Engineering System

GPES is a computer system which enables the rapid production of user oriented computer aided design systems for engineering of chemical

processes. In using GPES to create a computer aided design system, one has to define different types of process elements (process units, streams, etc.) which may be present in the flowsheets to be analyzed or designed by the users of that system. The subroutines performing the computations for these elements (process moduels, etc.) must also be provided.

The different types of process elements are defined by providing an information set for each of them. Each such an information set is called a template in GPES terminology. For example a template for a specific type of a process unit contains such information as the number of inlet and outlet streams, number of unit parameters, etc. Thus, a system created by GPES is called a Template Based System (TBS). The templates for each TBS reside on a set of files called template data base. Hence, the creation of a TBS consists of creation of a Template Data Base and development of a package of subroutines, mainly to represent the mathematical models of process units defined in that template data base. These subroutines are called TBS programs.

Each TBS is the responsibility of a system administrator. A TBS administrator may be assisted by a group of programmers for development of TBS programs. This group is known as TBS programmers. Once a TBS has been implemented (Templates have been defined and computational subroutines provided) a process designer (user) may use it to analyze any arbitrary specified configuraton of process units which are already defined in that TBS.

The structure of the organization of a team using GPES is shown in Figure 1.1. There are four levels of activity associated with GPES. Each level is the responsibility of a different set of personnel:

- 1) The GPES administrator who is responsible for:

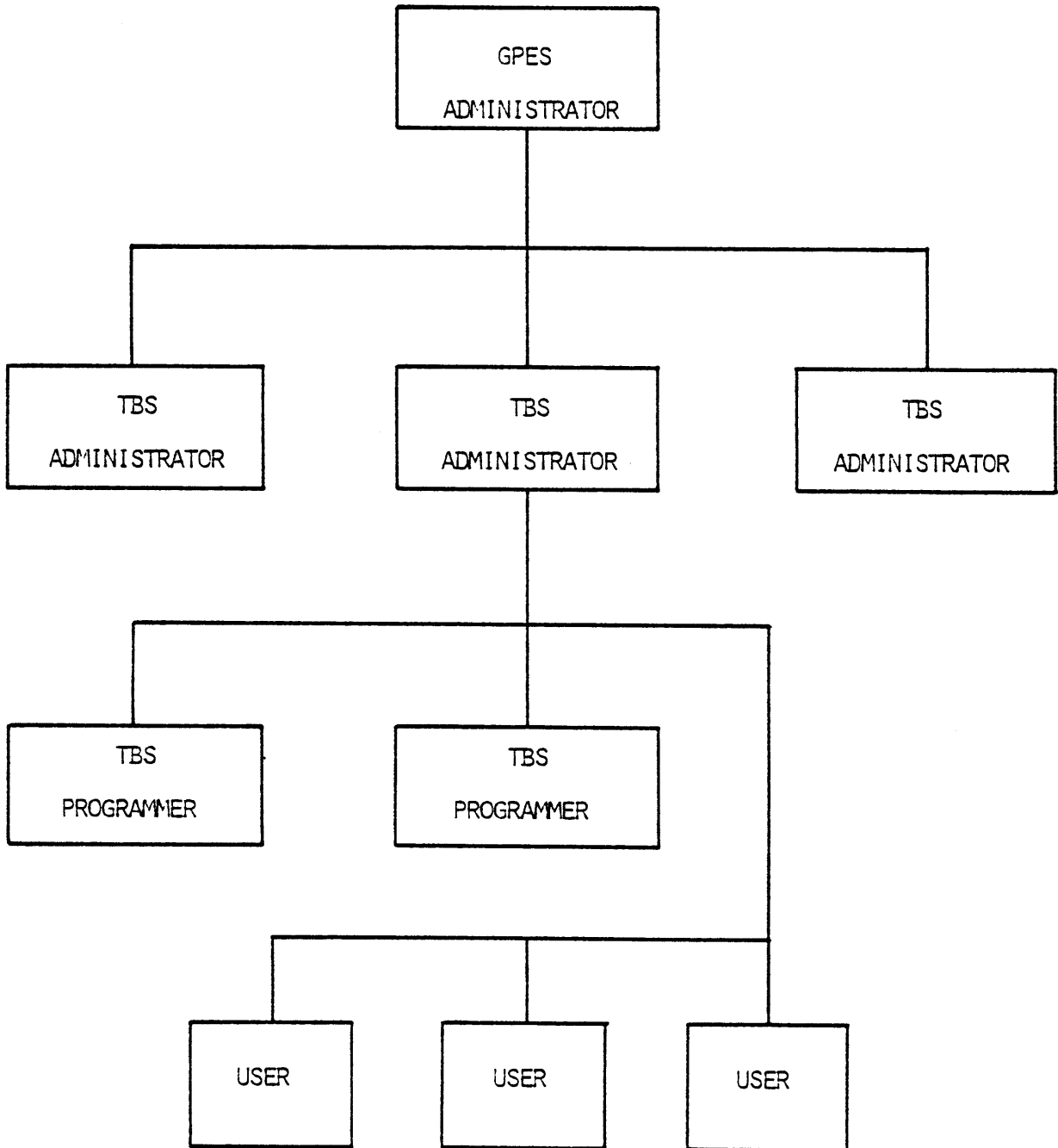


FIGURE 1.1 THE HIERARCHICAL STRUCTURE OF
GPES USAGE

- a) Maintenance of the GPES.
- b) Protecting the system from unauthorized TBS administrators.
- c) Protecting each TBS from Unauthorized users.

2) TBS administrators.

They are responsible for:

- a) Implementation and maintenance of Template Based Systems.
- b) Coordination of TBS Programmers.
- c) Permitting a user to use a TBS.

3) TBS programmers.

They are responsible for development of TBS programs. A TBS may contain the following programs:

- a) Routines performing calculations for individual process units (process modules).
- b) Physical and thermodynamic property estimation routines.
- c) Regression analysis programs.
- d) Other programs which may be required for a particular TBS.

4) Users.

They are the ultimate users of the Template Based Systems, the designers of chemical processes. A user may have access to one or more of these systems.

For each of these groups a set of tools and mechanisms (programs and languages used to communicate with those programs) has been developed to assist them in performing their responsibilities. GPES consists of these tools. The use of the system is shown in Figure 1.2.

A TBS Administrator creates and manipulates his template data base by an interactive program called "update_tdb". A language called Template Definition Language (TDL) has been provided to enable him to easily communicate with the "update_tdb" program to define his templates.

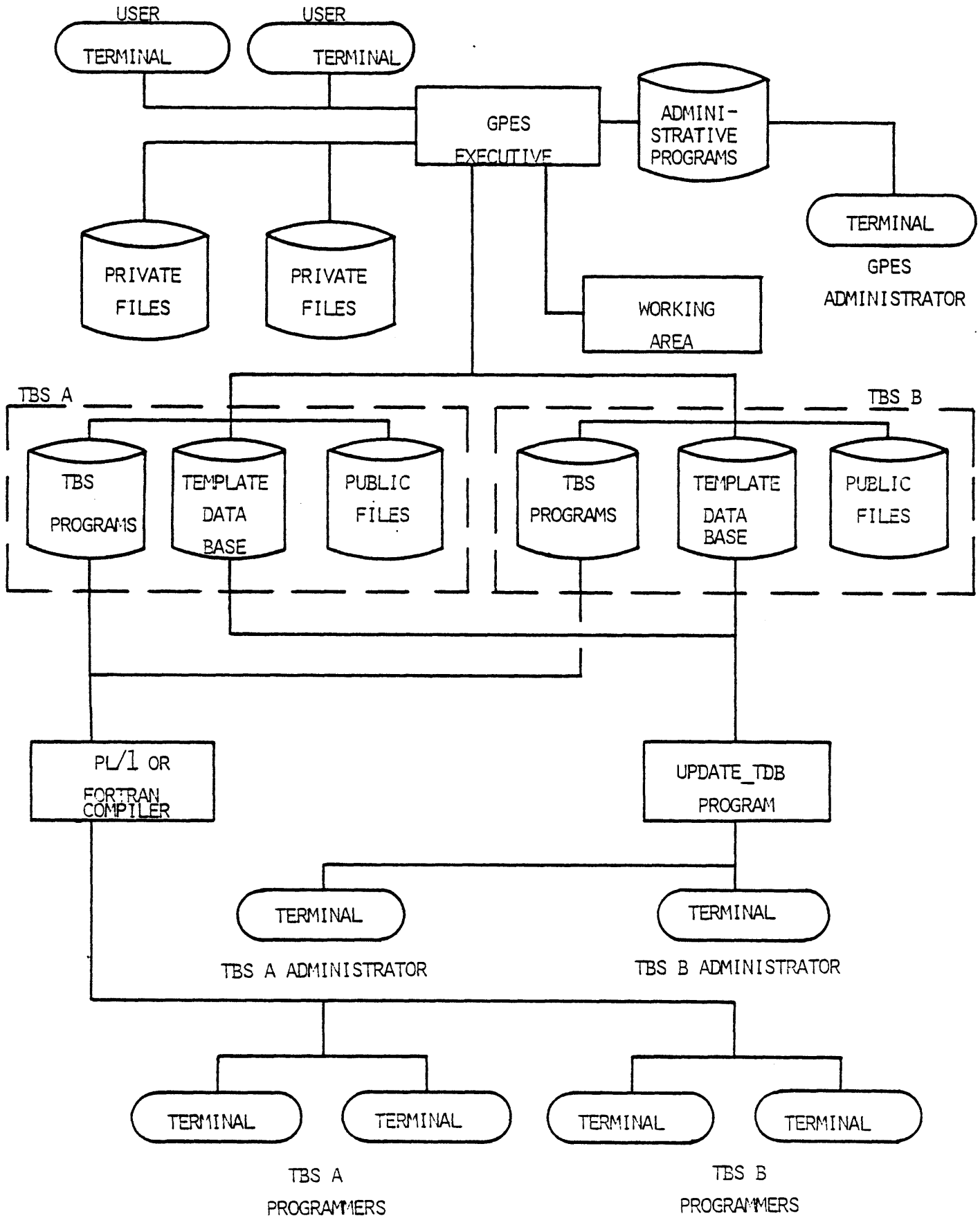


FIGURE 1.2 USE OF GPES

A TBS programmer writes the TBS programs in procedural languages such as PL/1 or Fortran. Development of these programs has been standardized and a package of service routines has been developed to assist him in his effort.

The backbone of the system is the executive program, which provides the means to execute the users' commands. In essence, the executive is a table driven interpreter, the tables being template data bases and GPES files. GPES files are part of a mechanism to protect the system from unauthorized TBS Administrators and to protect each TBS from unauthorized users.

GPES files contain information about each TBS and its authorized users. The GPES Administrator performs his administrative duties by creating and updating these files by an interactive program known as the administrative program. The executive program refers to these files to locate the template data base of a given TBS and to permit the user to use the specified TBS.

A special language has been designed to enable the user to easily communicate with the Executive. The language has been named PEL, which stands for Process Engineering Language. PEL consists of a series of commands. Each command is a request for an action to be taken by the system. The language enables a user to create and delete process elements, specify and unspecify parameters and variables of process elements, calculate (simulate) parts of or whole flowsheet and print results. PEL allows the choice of engineering units. Many other features are available within PEL.

All the information which is related to a particular flowsheet is represented by a network of data structures. This network is created and

manipulated by the Executive Program in response to the user commands. Changes in the flowsheet structure are reflected by changes in the network structure, as the initial concept of the design evolves into a final concept.

This network is located in an area of storage known as the working area. The working area automatically expands as the need for more space is recognized. Hence, there is no limitation on the size of the flowsheet being analyzed. The user can save this network in a file and later, retrieve it for further analysis.

This type of file is known as a process file. A user may have any number of process files each of which may contain any number of processes. The Executive Program creates and manipulates these files in response to the user's commands. The users may share their process files. This will promote team work on design and analysis of large process flowsheets. A user may also have any number of another type of file known as component files. A component file may contain the physical properties or constants for estimating these properties for any number of chemical components. The Executive Program also creates and manipulates these files in response to the user's commands. Hence, the user may not be explicitly concerned about the creation and maintenance of these files, outside the program. The users may also share their component files. A TBS administrator may create such files and make them accessible to all users of that TBS. Such files are known as public component files.

1.5.2 Development Process of a Template Based System

Development process of a TBS in general is shown in Figure 1.3. It consists of the following phases:

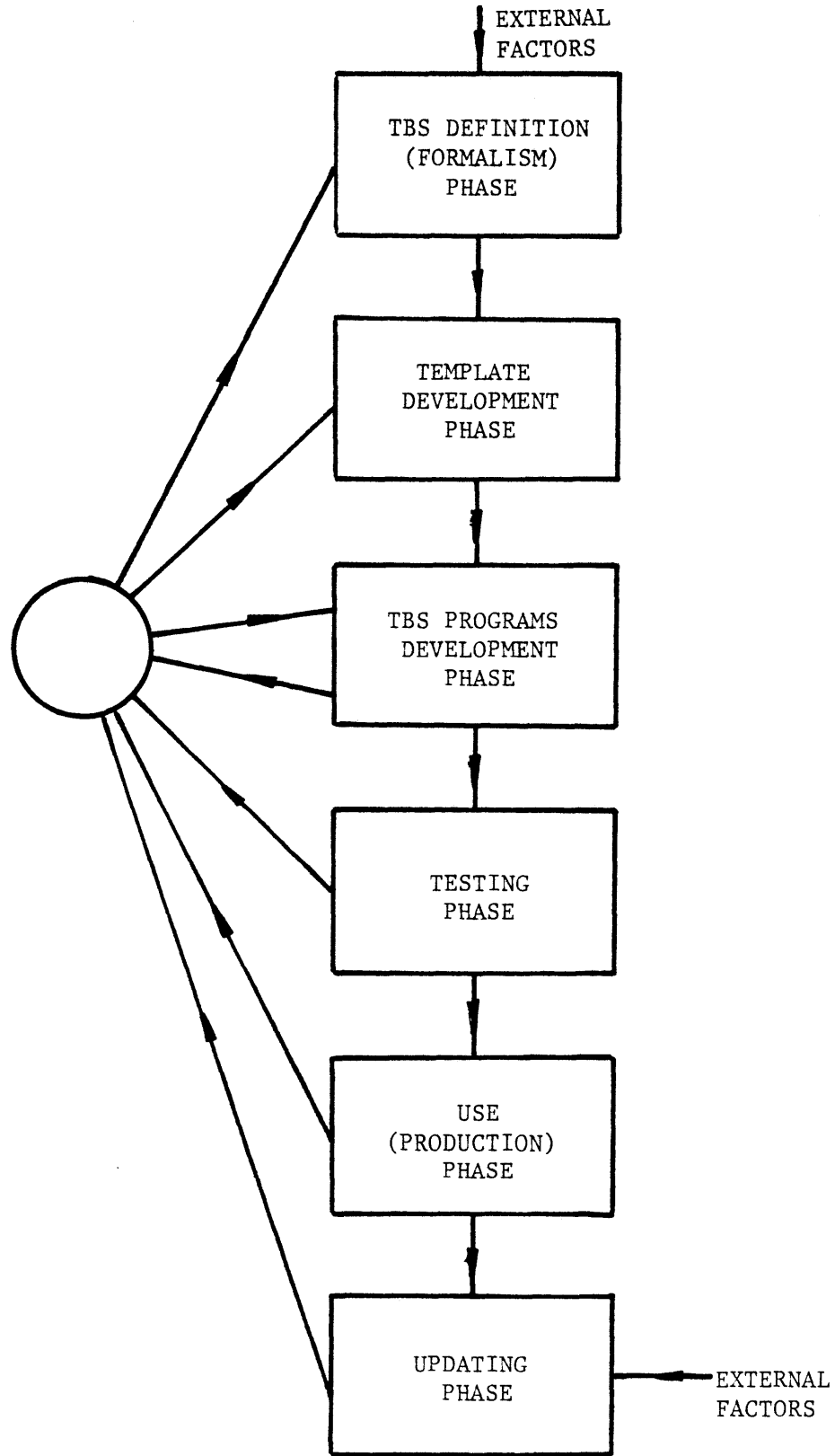


FIGURE 1.3 DEVELOPMENT PROCESS OF A TBS

1. TBS definition (Formalism) phase.

Once the need for development of a new TBS is brought about by external factors, this would be the first phase of development. In this phase the objective of the TBS should be stated and different types of process units and streams that may be required should be identified. Decisions should also be made mainly on how to represent the process units and streams. The mathematical models of the process units should also be prepared. This phase not only provides input to the following two phases, but it is essential for documentation purposes.

2. Template Definition Phase.

In this phase the TBS Administrator using the "update_tdb" program defines a template mainly for every unit type and stream type.

3. TBS Programs Development Phase.

In this phase the TBS programmers develop a subroutine for every unit type to represent its mathematical model. In this effort sometimes the need for updating or modification of efforts made in previous phases is realized, in which case the TBS administrator has to return to one of the previous phases.

4. Testing Phase.

Before the TBS is released to the public, the TBS Administrator should test it by simulating various process flowsheets. Once any error is found, he may have to return to one of the previous phases for debugging.

5. Use (Production) Phase.

At this phase the TBS may be used by users for analysis and design of various process flowsheets. In the course of the TBS usage,

the users may discover some errors in the TBS or may recognize the need for expansion or improvement of the TBS. The users should inform the TBS Administrator of the need for modification.

6. Updating Phase.

A TBS is an open-ended system which can be easily extended or modified. The need for extension or modification may be realized by inputs received from the previous phase or by other external factors. This will lead the TBS administrator to one of the first three phases for extending or modifying the TBS.

1.5.3 Development of a Number of Prototype Template Based Systems

Using GPES three prototype TBS's have been created to demonstrate the application and use of GPES and systems created by GPES. These prototype TBS's are as following:

1. Heat Exchanger Networks Analyzer. This prototype TBS is capable of analyzing heat exchanger networks. It has been created to demonstrate that using GPES, one could develop a very simple system to solve a particular class of problems.
2. TBS-II. This prototype TBS is capable of analyzing processes with conventional liquid-vapor streams, particularly hydrocarbon processes. Currently it contains a few types of process units such as distillation column, heat-exchanger, and isothermal flash separation, and hence it is limited to processes having only these process elements.
3. MHD. This prototype TBS is created to analyze MHD processes. It is an example of a case where existing simulators can not be used to simulate processes having streams other than conventional liquid and vapor streams.

It should be noted that any of the above three TBS's could be expanded to include additional unit operations. It is also not necessary to have different TBS's to analyze different classes of processes. In fact, a single TBS could have been created capable of analyzing all three types of processes mentioned above.

1.5.4 An Example: Development of a TBS for Analyzing Heat Exchanger Networks

One of the prototype TBS's which has been created is for analysis of heat exchanger networks. To demonstrate the use and application of GPES and systems created by GPES, the development process of this simple TBS is briefly described next. It is presented in the framework described earlier.

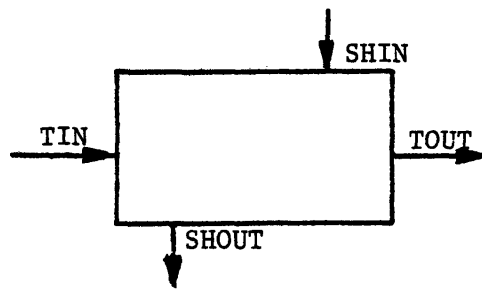
1. TBS Definition Phase (formalism).

The objective of this TBS is to analyze the steady state behavior of an arbitrary heat exchanger network.

Each stream in a heat exchanger network is characterized by two variables: the product of mass flow rate and specific heat (WC) and the Temperature (T). Pressure and composition are not considered to be pertinent variables.

There are four types of units: counter-current heat exchanger, mixer, divider, and convergence unit.

Counter-current heat exchanger is modeled as shown in Figure 1.4. Unit type mixer is for adding two streams and is modeled as shown in Figure 1.5. Unit type divider is for splitting one stream into two streams and is modeled as shown in Figure 1.6. Unit type convergence is for testing and promoting convergence for a recycle stream. Most chemical processes involve recycle streams.

Specifications:

<u>Parameters:</u>	U	overall heat-transfer coefficient (e.g., Btu/hr-ft ² -of)
	A	area (e.g., ft ²)
<u>Connections:</u>	TIN	tube inlet
	TOUT	tube outlet
	SHIN	shell inlet
	SHOUT	shell outlet

Equations:

Material Balances:

$$WC_{TOUT} = WC_{TIN}$$

$$WC_{SHOUT} = WC_{SHIN}$$

Energy Balances:

$$T_{TOUT} = \frac{(1 - R)T_{TIN} + R(1 - F)T_{SHIN}}{1 - RF}$$

$$T_{SHOUT} = T_{SHIN} - \frac{T_{TOUT} - T_{TIN}}{R}$$

where:

$$R = \frac{WC_{SHIN}}{WC_{TIN}} ; \quad F = \exp \frac{UA}{WC_{SHIN}} (R - 1)$$

Special case (R = 1):

$$T_{TOUT} = \frac{T_{TIN} + \alpha T_{SHIN}}{1 + \alpha},$$

with $\alpha = UA/WC_{SHIN}$

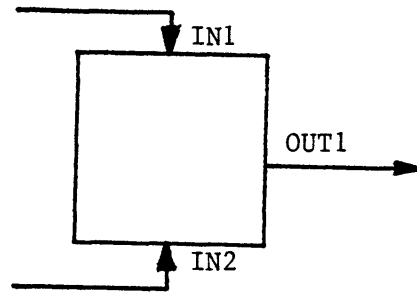
Input Variables

WC_{TIN}, T_{TIN}
 WC_{SHIN}, T_{SHIN}
 U, A

Output Variables

WC_{TOUT}, T_{TOUT}
 WC_{SHOUT}, T_{SHOUT}

Figure 1.4 Heat Exchangers TBS - Countercurrent Exchanger Model



Specifications:

<u>Parameters:</u>	None
<u>Connections:</u>	IN1 first inlet stream
	IN2 second inlet stream
	OUT1 outlet stream

Equations:

Material Balance: $WC_{OUT1} = WC_{IN1} + WC_{IN2}$

Energy Balance:

$$T_{OUT1} = \frac{WC_{IN1} T_{IN1} + WC_{IN2} T_{IN2}}{WC_{IN1} + WC_{IN2}}$$

Input Variables

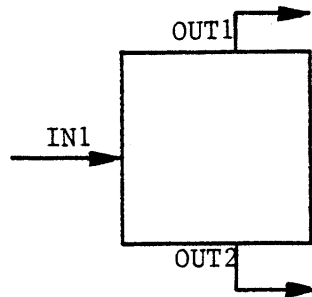
WC_{IN1}, T_{IN1}

WC_{IN2}, T_{IN2}

Output Variables

WC_{OUT1}, T_{OUT1}

Figure 1.5 Heat Exchangers TBS - Mixer Model



Specifications:

<u>Parameters:</u>	F	Fraction of inlet stream, IN1, diverted to outlet stream OUT1
<u>Connections:</u>	IN1	inlet stream
	OUT1	first outlet stream
	OUT2	second outlet stream

Equations:

Material Balances:

$$WC_{OUT1} = F WC_{IN1}$$

$$WC_{OUT2} = (1 - F) WC_{IN1}$$

Energy Balances:

$$T_{OUT1} = T_{IN1}$$

$$T_{OUT2} = T_{IN1}$$

Input Variables

F
 WC_{IN1}, T_{IN1}

Output Variables

WC_{OUT1}, T_{OUT1}
 WC_{OUT2}, T_{OUT2}

Figure 1.6 Heat-Exchangers TBS - Divider Model.

Therefore, such processes contain information recycle loops. That is, cycles for which insufficient information is available to permit equations for each unit to be solved independently. The equations for units in an information recycle loop must be solved simultaneously. One solution technique is to "tear" one stream in the recycle loop [11,22,51,148]; that is, to guess variables of that stream. Based upon tear stream guesses, information is passed from unit to unit until new variables of the tear stream are computed. These new values are used to repeat the calculations until convergence tolerances are satisfied. Unit type convergence is used for comparing newly computed variables (feed stream to the convergence unit) with guess values (product stream from the convergence unit) and to compute new guess values when convergence tolerances (unit parameters) are not satisfied. The convergence unit is modeled as shown in Figure 1.7.

Physical dimensions pertinent to this TBS are as follows:

	<u>Standard Units</u>	<u>Optional Units</u>
1. Temperature	Degree F	Degree R, C, K
2. Area	FT ²	
3. Heat Rate	BTU/HR	
4. Heat Transfer Coefficient	BTU/HR-FT ² -F	

2. Template Definition Phase.

Using the "update_tdb" Program, the following templates have been defined:

- a. A template for the only existing stream type, std.
- b. A template for every one of the unit types: heatex, divider, mixer, and convergence.



Specifications:

Parameters:

MAXIT	Maximum number of iterations
NIT	Number of iterations
RDEV_WC	Relative deviation for stream parameter WC
ADEV_WC	Absolute deviation for stream parameter WC
RDEV_T	Relative deviation for stream parameter T
ADEV_T	Absolute deviation for stream parameter T
FLAG	Flag indicating convergence, it is positive if convergence has been achieved, it is negative otherwise.

Connections:

IN	Inlet stream
OUT	Outlet stream

Equations:

Test for convergence:

$$\text{If } |WC_{IN} - WC_{OUT}| \leq (RDEV_WC)(WC_{IN}) + ADEV_WC$$

$$\text{and } |T_{IN} - T_{OUT}| \leq (RDEV_T)(T_{IN}) + ADEV_T$$

then FLAG = +1, otherwise

$$WC_{OUT} = WC_{IN}, T_{OUT} = T_{IN}, \text{ and FLAG} = -1.$$

Default Unit Parameters:

$$MAXIT = 50$$

$$NIT = 0$$

$$RDEV_WC = .01$$

$$RDEV_T = .01$$

$$ADEV_WC = .05$$

$$ADEV_T = .05$$

$$FLAG = -1.$$

Figure 1.7 Heat Exchangers TBS - Unit Convergence Model

- c. A template for every one of the four dimension types. The collection of these templates, which is referred to as a dimension table, enables the system to automatically convert the user-supplied data into standard units, if they are provided in other optional units.
- d. A template containing other miscellaneous information such as the TBS name, TBS administrator's name, etc.

The printout of the dimension table, the stream template and the template for unit type heatex are shown in Figure 1.8.

3. TBS Program Development Phase.

In this phase a subroutine for each process unit type is developed to represent its mathematical model. The names of these subroutines have been already supplied in the unit templates. The system will call upon these routines to solve equations for each unit. The subroutine for unit Heatex is listed in Figure 1.9.

4. Testing Phase.

In this phase the TBS has been tested by simulating various heat exchanger networks.

5. Use (Production) Phase.

Now the system is ready to be used by ultimate users of the TBS, process designers. To use the TBS they only have to know PEL (Process Engineering Language). The following example demonstrates the use of the TBS and PEL.

Example

There are a number of identical heat exchangers ($A = 20 \text{ FT}^2$, $U = 10 \text{ BTU/HR-FT}^2\text{-}^\circ\text{F}$), and a number of cold streams ($WC = 500 \text{ BTU/HR}$, $T = 300^\circ\text{F}$) to be used to cool a hot stream ($WC = 1000$

FIGURE 1.8 HEAT EXCHANGERS TBS - PRINTOUT OF SOME TEMPLATES

list units

```

UNIT TYPES
heatex
mixer
divider
convergence
ENTER COMMAND:
print dimtable
    
```

DIMENSIONS TABLE

NUMBER OF DIMENSION TYPES = 4

DIMENSION TYPE	NAME	STANDARD UNITS	OPTIONS	A	B
1	temperature	f	r	-4.6000000e+002	1.0000000e+000
			c	3.2000000e+001	1.8000000e+000
			k	-4.6000000e+002	1.8000000e+000
2	area	ft2			
3	heat_rate	btu/hr			
4	heat_transfer_co	btu/hr-ft2-f			

NOTE: DIMENSION TYPE OF A DIMENSION LESS PARAMETER IS ZERO.
 ENTER COMMAND:
 print stream std

STREAM TYPE = std
 REFERENCE = a standard stream type for this TBS
 TYPE OF COMPONENTS FLOWING IN THIS STREAM = none

NUMBER OF PHASES = 0

PHASE 0 (TOTAL STREAM)

NUMBER OF PHASE PARAMETERS = 2

PARAMETER	DIMENSION TYPE
1- wc	3
2- t	1

NUMBER OF FLOW PARAMETERS = 0

PROCEDURE TO CALCULATE THE STREAM = ;

FIGURE 1.8 CONTINUED

```
print unit heatex
```

```
UNIT TYPE = heatex
REFERENCE = counter_current heat exchanger.
```

```
NUMBER OF UNIT PARAMETERS =      2
```

PARAMETER	DIMENSION TYPE
1- u	4
2- a	2

```
NUMBER OF INLETS =      2
```

INLET CONNECTIONS	STREAM TYPE
1- tin	std
2- shin	std

```
NUMBER OF OUTLETS =      2
```

OUTLET CONNECTIONS	STREAM TYPE
3- tout	std
4- shout	std

```
PROCEDURE TO CALCULATE THE UNIT = heatex
MINIMUM NO OF ARGUMENTS =      0
MAXIMUM NO OF ARGUMENTS =      1
```

```
NUMBER OF LEVELS OF CALCULATION =      1
```

```
VALUE STATUS CODES FOR LEVEL =      1
```

```
UNIT PARAMETERS
```

PARAMETER	CODE
1- u	7
2- a	7

```
FOR CONNECTION = tin
```

```
THIS CONNECTION IS REQUIRED
```

```
STREAM CONNECTED TO THIS CONNECTION = std
```

```
PHASE 0 (TOTAL STREAM)
```

PHASE PARAMETER	CODE
1- wc	7
2- t	7

```
FOR CONNECTION = shin
```

```
THIS CONNECTION IS REQUIRED
```

```
STREAM CONNECTED TO THIS CONNECTION = std
```

```
PHASE 0 (TOTAL STREAM)
```

PHASE PARAMETER	CODE
1- wc	7
2- t	7

```
FOR CONNECTION = tout
```

```
THIS CONNECTION IS REQUIRED
```

```
STREAM CONNECTED TO THIS CONNECTION = std
```

```
PHASE 0 (TOTAL STREAM)
```

PHASE PARAMETER	CODE
1- wc	13
2- t	13

```
FOR CONNECTION = shout
```

```
THIS CONNECTION IS REQUIRED
```

```
STREAM CONNECTED TO THIS CONNECTION = std
```

```
PHASE 0 (TOTAL STREAM)
```

PHASE PARAMETER	CODE
1- wc	13
2- t	13

```
ENTER COMMAND:
```

FIGURE 1.9 HEAT EXCHANGERS TBS - HEATEX CALCULATING ROUTINE

heatex.pl1 05/07/78 1541.1 edt Sun

```

heatex|Proc(Punit,para,switch,error_switch)|
      dcl punit_ptr,
          para_ptr,
          switch bit(1),
          error_switch bit(1)|
      dcl unit_ptr entry(ptr,fixed bin,ptr,bit(1))|
      dcl strm_ptr entry(ptr,fixed bin,ptr,bit(1))|
      dcl xput_parm entry(ptr,fixed bin,float bin,bit(1))|
      dcl xset_parm entry(ptr,fixed bin,float bin,fixed bin,bit(1))|
      dcl (pparm,pstream) ptr|
      dcl exp builtin|
      dcl vtype,
          code bit(1)|
      dcl (u,a,ttin,ttout,tshin,tshout,wctin,wctout,wcshin,wcshout,r,f) float bin|

/* retrieve input variables */
      call unit_ptr(punit,0,pparm,code)|
      call xset_parm(pparm,1,u,vtype,code)|
      call xset_parm(pparm,2,a,vtype,code)|
      call unit_ptr(punit,1,pstream,code)|
      call strm_ptr(pstream,0,pparm,code)|
      call xset_parm(pparm,1,wctin,vtype,code)|
      call xset_parm(pparm,2,ttin,vtype,code)|
      call unit_ptr(punit,2,pstream,code)|
      call strm_ptr(pstream,0,pparm,code)|
      call xset_parm(pparm,1,wcshin,vtype,code)|
      call xset_parm(pparm,2,tshin,vtype,code)|

/* Perform required computations */

/* material balance */
      wctout=wctin|
      wcshout=wcshin|

/* energy balance */
      r=wcshin/wctin|
      f=exp(u*a/wcshin*(r-1))|
      if r=1
          then ttout=(ttin*u*a/wcshin*tshin)/(1+u*a/wcshin)|
          else ttout=((1-r)*ttin+r*(1-f)*tshin)/(1-r*f)|
      tshout=tshin-(ttout-ttin)/r|

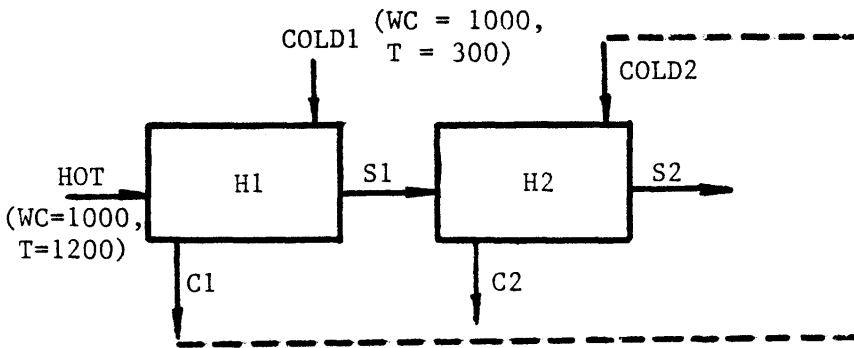
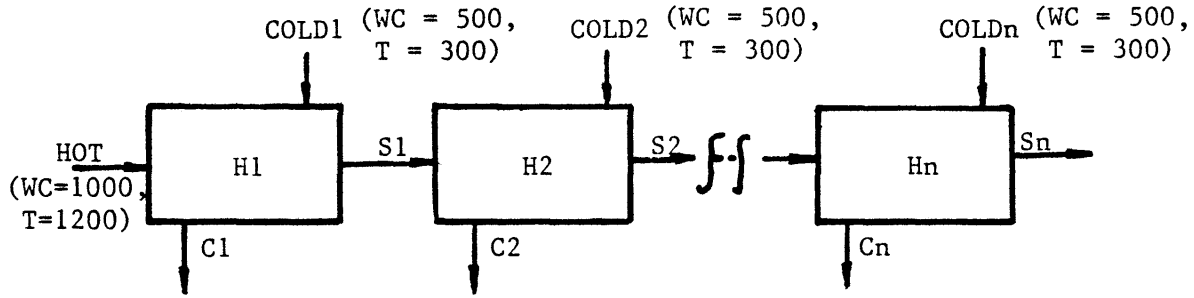
/* store output variables */
      call unit_ptr(punit,3,pstream,code)|
      call strm_ptr(pstream,0,pparm,code)|
      call xput_parm(pparm,1,wctout,code)|
      call xput_parm(pparm,2,ttout,code)|
      call unit_ptr(punit,4,pstream,code)|
      call strm_ptr(pstream,0,pparm,code)|
      call xput_parm(pparm,1,wcshout,code)|
      call xput_parm(pparm,2,tshout,code)|

      return|
end heatex|

```

BTU/HR, $T = 1200^{\circ}\text{F}$) to below 950°F , as shown in Figure 1.10a. The problem is determining the minimum number of required heat exchanges. The PEL computer session for solving this problem is shown in Figure 1.11. To solve this problem, one may start with one heat exchanger and increase the number of heat exchangers until the hot stream is cooled to the desired temperature. As can be seen in Figure 1.11, two heat exchangers will bring the hot stream temperature below the 950°F . Before terminating the session, the user has saved his process network, so that he may be able to continue his design effort sometime in the future.

At a later session, the user wishes to investigate other design alternatives, especially those shown in Figures 1.10b and 1.10d. The PEL computer session is demonstrated in Figure 1.12. As can be seen, the output temperature for process configuration shown in Figure 1.10b is 950°F compared to 945°F for the first configuration. To simulate the process flowsheet shown in Figure 1.10c, one observes that unit H1 and H2 cannot be calculated independently. To calculate unit H1 stream cold1 should be known. To determine stream cold1 unit H2 should be calculated which requires the stream S1 to be known. Therefore, to calculate unit H1 stream S1 should be known, but on the other hand, to find the stream S1 requires the calculation of Unit H1. Thus, it can be seen that the process flow sheet contains an information recycle loop; that is, too few stream variables are known to permit equations for each unit to be solved independently. One solution technique is to tear one stream in the recycle loop; that is, to guess variables of that stream. Based upon tear stream guesses,



NOTES

WC IN BTU/HR
T IN °F

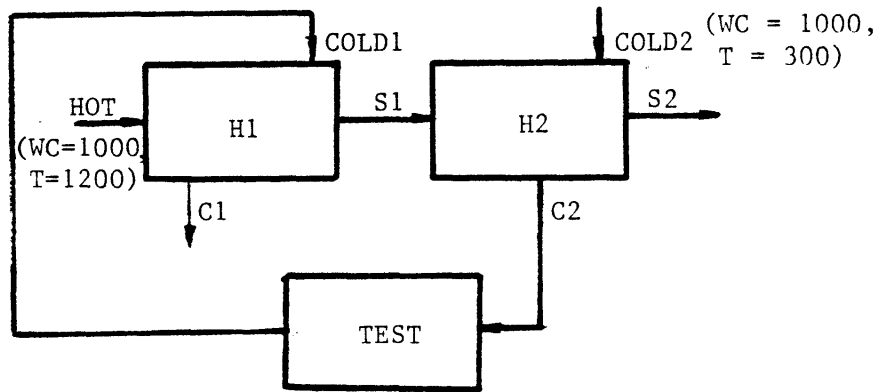
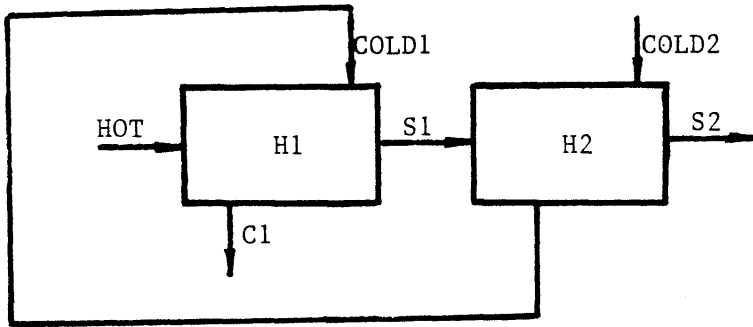


FIGURE 1.10 HEAT EXCHANGERS TBS - THE PROCESS FLOWSHEET FOR THE EXAMPLE CASE

FIGURE 1.11 HEAT EXCHANGERS TBS - A PEL COMPUTER SESSION FOR THE EXAMPLE CASE

pel brief

```

*beginning of attachment process.
**enter the name of TBS you wish to use now:heat_exchangers

*attachment process has been successful.

*new process created at:05/07/78 1618.5 edt Sun
**enter maximum number of components(0 to 200):0

**COMMAND :      $---DESCRIBE THE PROCESS CONFIGURATION---$

**continue:create unit(h1) type=heatex;

**COMMAND :create streams(hot,cold1,s1,c1);

**COMMAND :connect unit h1 at tin=hot shin=cold1 tout=s1 shout=c1;

**COMMAND :      $---SPECIFY KNOWN PARAMETERS---$

**continue:specify unit(h1) (a=20'ft2', u=10);

**COMMAND :specify stream(hot) (wc=1000'btu/hr' , t=1200'f');

**COMMAND :specify stream(cold1) (wc=500 , t=300);

**COMMAND :      $---SIMULATE THE PROCESS AND PRINT THE RESULT---$

**continue:calculate unit(h1);

*entering routine heatex          for level      1 calculation of unit      'h1'
**COMMAND :print variable(s.s1.p0.t);

          s.s1.p0.t=                      1.061892e+003 f          calculated
**COMMAND :

**continue:      $expand the process configuration$

**continue:let unit h2=h1;

**COMMAND :let stream cold2=cold1;

**COMMAND :connect unit h2 at all=s1,cold2,s2,c2;

*stream 's2' does not exist. a stream of type 'std' has been created.
*stream 'c2' does not exist. a stream of type 'std' has been created.
**COMMAND :      $---SIMULATE AND PRINT THE RESULT---$

**continue:calc unit(h2);

*entering routine heatex          for level      1 calculation of unit      'h2'
**COMMAND :print v(s.s2.p0.t);

          s.s2.p0.t=                      9.449777e+002 f          calculated
**COMMAND :      $---SAVE THE PROCESS---$

**continue:save process net1;

***ERROR*** s 91 no process file is opened.
*command ignored.
**COMMAND :      $---CREATE AND OPEN A PROCESS FILE---$

**continue:open process file(demo);

**enter the relative or absolute pathname of the directory containing the process file 'demo'
(if it is the same as your working directory ,>udd>ICPES>Arab-Ismaili, enter a null line):

*process file 'demo' does not exist.
**if you wish a new one be created enter yes ,otherwise no:yes

*process file 'demo' is opened.
**COMMAND :save process net1;

*process has been saved.
**COMMAND :end;

*thank you Arab-Ismaili for trying GPES come back soon, bye!
r 1638 6.163 271.965 5339

```

FIGURE 1.12 HEAT EXCHANGERS TBS - ANOTHER PEL COMPUTER SESSION FOR THE EXAMPLE CASE

pel brief

```

*beginning of attachment process.
**enter the name of TBS you wish to use now:heat_exchangers

*attachment process has been successful.

*new process created at:05/07/78 1648.9 edt Sun
**enter maximum number of components(0 to 200):0

**COMMAND :          $---LOAD THE PROCESS---$

**continue:open process file(demo);

**enter the relative or absolute pathname of the directory containing the process file 'demo'
(if it is the same as your working directory ,>udd>ICPES>Arab-Ismaili, enter a null line):

*process file 'demo' is opened.
**COMMAND :load process net1;

*process 'net1' has been created at:05/07/78 1618.5 edt Sun
  by system: GPES          serial_no: 1 compet_level: 1
  and TBS: heat_exchangers serial_no: 1 compet_level: 1
  and it has not been accessed by any other incompatible system since.
*process has been loaded.
**COMMAND :          $---SIMULATE THE PROCESS FOR CONFIGURATION SHOWN IN FIGURE 1.10b ---$

**continue:specify stream (cold1)(wc=1000);

**COMMAND :calculate unit(h1);

*entering routine hestex          for level 1 calculation of unit 'h1'
**COMMAND :let stream cold2=c1;

**COMMAND :calc unit(h2);

*entering routine hestex          for level 1 calculation of unit 'h2'
**COMMAND :print variable(s.s2.p0.t);

          s.s2.p0.t=          9.500000e+002 f          calculated
**COMMAND :

**continue:          $---SIMULATE THE PROCESS FOR CONFIGURATION SHOWN IN FIGURE 1.10 d ---$

**continue:create unit(test) type=convergence;

**COMMAND :connect unit test at in=c2 out=cold1;

**COMMAND :          $---use the default values for parameters of unit test---$

**continue:calc u(test(2));

*entering routine convs          for level 2 calculation of unit 'test'
**COMMAND :specify stream(cold2)(wc=1000,t=300);

**COMMAND :          $---assume the initial values of test stream---$

-----
**continue:assume stream(cold1) (wc=1000,t=400);

**COMMAND :calculate units(h1,h2,test,(h1));

*entering routine hestex          for level 1 calculation of unit 'h1'
*entering routine hestex          for level 1 calculation of unit 'h2'
*entering routine convs          for level 1 calculation of unit 'test'
*entering routine hestex          for level 1 calculation of unit 'h1'
*entering routine hestex          for level 1 calculation of unit 'h2'
*entering routine convs          for level 1 calculation of unit 'test'
**COMMAND :          $---as can be seen convergence has been achieved in two iterations---$

**continue:          $---print the result---$

**continue:print variable(s.s2.p0.t);

          s.s2.p0.t=          9.427469e+002 f          calculated
**COMMAND :          $---SAVE THE NEW CONFIGURATION FOR FUTURE STUDIES---$

**continue:save process net1 override;

*process has been saved.
**COMMAND :end;

*thank you Arab-Ismaili for trying GPES come back soon, bye!

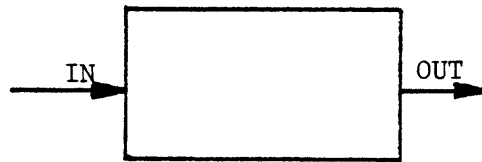
```

information is passed from unit to unit until new values of the variables of the tear stream are computed. These new values are used to repeat the calculations until convergence tolerances are satisfied. This has been done as shown in Figure 1.10d and demonstrated in Figure 1.12.

As can be seen, convergence has been achieved in two iterations. The output temperature is 943°F which is lower than the two previous outcomes.

6. Updating Phase (TBS Administrator's Task).

A TBS is an open ended system which can easily be extended or modified. Suppose a new unit type is to be added to the TBS. The unit type is adjuster, which heats or cools a stream to a specified temperature. The unit model is shown in Figure 1.13. A new template defining this new unit type has been added to the template based system and a subroutine representing the mathematical model of the unit has been developed, and the updated TBS has been tested. Now the users of the TBS may incorporate units of this type in their process flowsheets when such a need arises.



Specifications:

Parameters: Q Heat added (removed) (e.g., Btu/hr)

Connections: IN Inlet stream
 OUT Outlet stream

Equations:

Material Balance: $WC_{OUT} = WC_{IN}$

Energy Balance: $Q = (T_{OUT} - T_{IN})WC_{IN}$

Input Variables

WC_{IN}

T_{IN}

T_{OUT}

Output Variables

WC_{OUT}

Q

Figure 1.13 Heat Exchangers TBS - Adjuster Model

1.6 CONCLUSIONS AND THESIS CONTRIBUTIONS

There are two basic problems with the existing general purpose process simulators:

- a) The existing systems are applicable to those process flowsheets having only conventional vapor-liquid streams. This inflexibility makes it either impossible or very difficult to expand present-day systems to include other types of processes such as coal processing or electric power generating systems.
- b) The existing systems are mostly developed for simulation and do not provide the design atmosphere for process engineers. A general purpose simulator, to be effective as a design tool, must use a mode of operation, methods of input and output, and calculation techniques that minimize the effort required for designer-computer communication so as to maximize effective interaction between the designer and the computer. The time scale is important in process design.

The objective of this thesis was to develop a framework for the development of general purpose chemical process simulators that:

- a) are applicable to all types of chemical processes, and
- b) are more adaptable to the design environment.

In fulfillment of the above objective, the thesis has provided the following contributions in the field of computer aided design for process engineering:

1) Formulation of a general model for chemical processes, to represent the process flowsheet for any level of sophistication that may be used to analyze that process. The introduction of the concept of a general stream and flow parameters have been the major contributions for achieving this result.

2) Identification of the issues involved in the design of a computer system for process engineering. The work has distinguished two classes of problems:

- a) Those that are common to the design of any system,
- b) Those that are related to a particular system.

3) Based upon the above findings, the work has led to the design and implementation of a General Process Engineering System (GPES) which allows any group or organization to easily and systematically build its own system. These systems could be very simple or very sophisticated depending on the particular needs and applications of the group.

The design and implementation of the GPES as a tool for creating computer aided design systems for chemical process engineering provides the following advantages:

- a) Allows any group or organization to easily and systematically build its own design system, thereby reducing the time and effort required to produce such systems.
- b) The created system may not be limited to a particular class of processes. Such a system is open-ended and capable of analyzing any type of process.
- c) The created system is more adaptable to the design environment, and provides the user with features not usually found in existing general purpose simulators.

In summary, a system created by GPES (a TBS) has the following characteristics:

- a) Flexible. It is applicable for analyzing any type of process flowsheets. It is not limited to process flowsheets having only conventional vapor-liquid streams. The system can be easily modified, expanded and updated.
- b) The user communicates with the system by a problem oriented language (PEL). The user may enter his input data in any allowable units of measurement. Arithmetic expressions may be used where numerical data is expected. The system performs extensive types of error checking such as detection of over- or under-specification of process units and streams. The system produces over 200 easy-to-understand messages to detect the user's negligence.
- c) Interactive. Allowing the progressive design of a process.
- d) Integrated. Capable of the following functions:
 - i) Simulating or designing a process for any required level of sophistication such as: preliminary process feasibility studies, plant design, equipment sizing, plant modifications, debottlenecking studies, effects of operational changes on plants, contractor checkout. The appropriate programs should be provided in that TBS.
 - ii) Serving as a physical property data base system. It also promotes sharing of data among users.

- iii) Analyzing experimental data (regression analysis).
 - iv) Serving as a general purpose interpreter.
 - v) Serving as a desk calculator for arithmetic operations.
- e) Dynamic creation and modification of process flowsheet.
Enabling the user to instruct the computer to alter process configuration or operating parameters without having to redescribe the problem.
- f) The ability to save and retrieve user process models. This will save both designer and computer time in not having to reinitiate the problem. It also promotes sharing of process models among users, and enhances teamwork.
- g. It provides virtual memory. There is no limitation on the size of the process flowsheet being analyzed by the user.
- h. Ease of use in that no knowledge of programming and job control language is required. All that is required is knowledge of PEL (Process Engineering Language).
- 4) The work has provided a comprehensive example of the use of current systems programming techniques (structured programming, dynamic storage allocation, manipulation of arbitrary data structures, list processing, etc.) and current computer technology (time sharing, virtual memory, dynamic linking and loading) in a systems programming application of interest to chemical engineering.

Using the GPES several prototype template based systems have been created. The results of these efforts indicated that:

- a) The GPES allows the creation of computer systems for different types of processes.
- b) The creation of such systems is simple and systematic.
- c) The features provided by the system are very useful and desirable for simulation and design of chemical processes.

In conclusion, the above studies and findings, and the availability of such a GPES, will benefit the following groups:

- a) Those interested in computer aided design for chemical engineering applications.
- b) Those planning to implement their own computer aided design systems for chemical processes (by reducing the characteristic 20-100 man-years effort formerly required to produce such systems). It allows them to focus their effort on the chemical engineering side of the problem, which results in the development of better process modules and comprehensive physical and thermodynamic property calculation packages.
- c) The process designers, by providing the ideal creative environment for process design by computer.
- d) Chemical engineering students in process design courses. By allowing them to implement their own system or to use an already developed educational one. They would be able to study whole processes as carefully as we now study individual unit operations.

CHAPTER 2

INTRODUCTION

2.1 Chemical Process Simulation

The simulation of chemical processing systems and the design of computing systems that can be utilized for such simulations has been an area of tremendous activity ever since the emergence of the general purpose digital computer as a design tool for engineers. Process simulation, in recent years, besides being accepted as a tool for the design and understanding of chemical processes, has also become an important requirement in the education of chemical engineers.

The simulation of any chemical process begins with the representation of the process by a mathematical model. This model is then solved either manually or, preferably, through the use of a computing aid to obtain information about the performance of the process under a given set of conditions. In most instances, the equations that constitute the mathematical model of the process, under steady state, are numerous and highly non-linear. The use of a digital computer for their solution becomes almost mandatory.

2.2 Simulation Versus Design

Two questions are normally asked concerning chemical processes. The first question is, given a specified flowsheet, how will it respond under given perturbations in stream variables or process unit parameters. This is normally called the simulation problem. It is characterized as the analysis of a fixed network of modules in a fixed configuration. Much time has been devoted to developing procedures or methodology for solving problems of this type. The results of these efforts have led to rules to govern simulation procedures. These rules have been incorporated into

computer systems which we shall classify as general purpose process simulators. A host of these simulators are in existence which have characteristically required tens of man-years to implement. A review of the state of the art is presented in Appendix A. The second question normally asked concerning chemical processes is, given a specified response of items in out-flowing streams from a process and given specified constraints on the parameters of the process units or streams within the process, which flowsheet will behave in the appropriate manner. This is called the design problem. The design problem may have many solutions, but as the number of constraints on the system's behavior is increased, the number of solutions is diminished until there may be no solutions. These problems are characterized by the fact that in arriving at a solution, not only process units must be selected, but the configuration of the process units must also be selected. Methodology for selecting process units and unit arrangement is not yet formalized, although efforts have been devoted to developing General Purpose Process Synthesizers[134] since the early 1970's. At present, computer-aided design systems must include man in the loop to carry out the important task of process unit selection and arrangement to accomplish a stated objective. It has not yet been established that these "creative" aspects of design, much of which are developed through experience (design know-how), can be formalized to the extent that they can be incorporated in a computer system which would eliminate man from the loop. Serious doubts are entertained as to whether this accomplishment is at all possible without subjecting the computer to the impossible task of analyzing the doubly infinite set of considering all possible process unit selections and configurations. Thus, the current procedure to solve design problems is for man to propose process unit

selection and arrangement, use a General Purpose Simulator to analyze the proposed network, and then compare the proposed network's response to the desired response and alter either network configuration or process unit selection, or both, until a network is developed with the desired characteristics. The process of design is necessarily iterative and having arrived at a solution, one is usually not certain that other solutions do not exist.

Considering the iterative nature of design problems, a general purpose simulator, to be effective as a design tool, must use a mode of operation, methods of input and calculation techniques that minimize the effort required for designer-computer communication so as to maximize effective interaction between the designer and the computer. The time scale is important in process design. A program with these characteristics can be used as an effective design tool and an efficient simulator.

2.3 Criteria for Computer Systems Amenable to Simulation and Design

There are two basic modes of computer operation, the off-line batch processing mode and the on-line interactive processing mode. In the former, the time between runs is long and this is acceptable when the results of one run are not critically important for the following run. However, in problems where, say, modifications to a design are dependent on results from a previous analysis, on-line processing is preferable. Moreover, the reduced waiting time enables the designer to remember what stage he had reached after the last processing and this continuity can often lead to higher efficiency.

On-line processing enables the designer to speed up the design process. However, replacing the batch mode by the interactive mode is not sufficient. The interface between the human mind and the computer must

also be modified to make the job easier for the designer, as discussed by Porter [133]. The essence of his discussion is that "it is necessary to create an atmosphere wherein the design team and the computer as an added partner are able to rapidly evaluate the effects of equipment arrangement and process variable selection in chemical process design".

He states that the answers to the following two questions are pertinent in determining the optimal use of the computer in such a design environment, to establish the specifications for a computer system and to establish the several unique concepts in programming that may be required to meet the specifications of the computer system.

One is, "How should the design activities be split between the designer and the computer?" The answer is fairly obvious. The designer's primary task is that of selecting and arranging equipment and evaluating the results of his efforts. The computer, therefore, should be used to carry out all the required analyses such as calculations of individual unit behavior and predictions of physical and thermodynamic properties required to perform these calculations. This is what most systems do.

The second question is, "How does the designer communicate with and maintain control over the computer?" This is the area where most automated design systems are inefficient. The designer is required to prepare some sort of input forms specifying the process configuration, the operating conditions of the units and the thermodynamic states of the streams. After the input is fed, the computer takes over complete control and arrives at a solution. Such a system is not very conducive to creative design. The environment of process design is much more dynamic. A designer rarely knows the entire plant configuration at the outset of a design. The flowsheet is more likely to evolve from an initial concept to

a final design after various perturbations of the initial concept. He must have more contact with and greater control over the computer. He must be allowed to specify what the computer should do at each stage of the design and receive feedback in terms of the intermediate results so that he can decide upon the next action to be taken by the computer.

One method of achieving this interaction is to provide a process design language which is more "natural" to the designer and can be interpreted by the computer. The language must be such that the designer can describe the flowsheet structure, request unit calculations, provide input, request output and have the computer carry out other instructions. Thus, he may solve a full flowsheet or any of its parts that he desires. He should also be able to change the flowsheet structure easily without respecifying the whole flowsheet.

The language should allow the choice of engineering units for input data to speed up the process of man-computer communication. The language should accept arithmetic expressions where numerical data are expected. The user should have access to every single piece of information about the process network and be able to refer to them in arithmetic expressions. Hence, the user will be able to relate different items of information about the process flowsheet. The system should allow the repetitive execution of a group of commands, to enhance the designer's iterative search. The system should allow the user to save the results of one analysis in a file to be used as input for further analysis. This will save both the designer and the computer time in not having to reinitiate the problem.

To perform process engineering effectively, the engineering data necessary for design must be made available. The computer can assist engineers in analyzing, estimating, and retrieving these data. However, it

is not efficient to have many individual computer-aided systems for simulation, design, physical property estimation, analyzing laboratory data, and so on. Therefore, such systems should be organized into an integrated system with a common data base, so that the consistency of these data are kept throughout the various stages of process engineering.

The description of a flowsheet should be independent of the programs for analysis. The same description should serve all analysis for which that flowsheet is applicable, for example, steady state simulation, equipment sizing, and economic evaluation, as noted by Evans et. al. [33], who discuss the requirements of an advanced computer system which will be needed to solve the process engineering problems of the 1980's. Such a system must be extendable and capable of modification. It should be easy to add new types of process units, to define new types of streams, species, etc. Hence, the system can be expanded to analyze new types of process. The inability of existing systems to analyze new energy conversion processes such as the magnetohydrodynamic (MHD) process has been due to the lack of this feature in these systems.

2.4 The Problem

There are two basic problems with the existing general purpose simulators:

- a. They are only applicable to those process flowsheets having conventional vapor-liquid streams. This inflexibility makes it either impossible or very difficult to expand present-day systems to include other types of process such as MHD, or other coal conversion processes.
- b. They do not meet the criteria discussed earlier, and, therefore, are not very useful as design tools.

The difficulties may be traced to following characteristics of existing programs:

- a. They are not interactive.
- b. They are not integrated.
- c. They do not provide natural problem oriented language.
- d. They do not have the capability of dynamic modification of the process network.
- e. They do not have the capability of storing the output model for future study.
- f. They are mostly developed for simulation, not for design.
- g. They are not flexible.

The implications of each shortcoming are examined next.

a. Existing Systems Are Not Interactive

The computer's role is as a tool that enables the designer to refine his understanding of a process by interactive analysis. Computer-aided design systems, to be effective, must permit the engineer to design processes in a dynamic environment. To use the existing systems, one prepares input forms, specifying process configuration, the operating conditions of units, the stream variables, etc. This input is fed to the program and the computer then takes over complete control, manipulating input data to arrive at a solution.

This mode of operation is not compatible with the dynamic environment of process design. The designer rarely knows the entire process configuration at the outset of a design. More usual is the gradual evolution of the total design starting with the initial concept of the plant and perturbing this concept by trying various process configurations until a final concept emerges. Therefore, the designer really requires

more intimate contact with and greater control over the computer's function, specifying at each stage of the design what the computer should perform, and, based on the results, deciding upon the next action to be taken by the computer.

b. The Existing Systems Are Not Integrated

Thermodynamic or transport properties that are not specified directly must be calculated. The computer should also assist the engineer in analyzing the laboratory data or in performing side calculations. However, it is not efficient to have many individual computer-aided systems for simulation, design, physical property estimation, analyzing the raw data, arithmetic computation, and so on. Therefore, such systems should be organized into an integrated system with a common data base, so that the consistency of these data are kept throughout the various stages of process engineering.

c. The Existing Systems Do Not Provide Natural Problem-Oriented Languages

Present systems generally require a rather rigid method of specifying the problem statement. The engineer should be able to communicate with the computer in terms that are familiar to him. Moreover, whatever language is provided, it must have the capability to transmit a wide variety of instructions to the computer.

d. Existing Systems Do Not Have the Capability of Dynamic Modification of Process Networks

In the dynamic environment of design, the engineer is often required to modify the process network. It is very desirable to be able to specify to the computer an alteration in process configuration or in operating parameters without having to redescribe the problem.

e. The Existing Systems Cannot Save The Results of Analysis

It is very useful to be able to save the results of analysis in a file to be used as input for further analysis. This will save both designer and computer time in not having to reinitiate the problem.

f. The Existing Systems are Mostly Developed for Simulation, Not for Design

For each process unit there is a set of equations of the form $f(\text{inlet-streams, operating conditions, outlet-streams}) = 0$. Simulation programs attempt to solve the equations in the form: $\text{outlet-streams} = f(\text{inlet-streams, operating conditions})$. Design programs must be capable of solving the former set of equations in its general form. However, a general purpose process simulator could be used as a trial-and-error tool to solve the design problem. Flowtran and Design systems provide feed-back and feed-forward control blocks which automate this trial-and-error process to some extent. But the control block technique is not applicable in all cases.

Although it is not feasible to develop process unit models capable of solving all possible forms of the design problem, models should be developed which are capable of solving a limited set of problems and the rest of the cases may be solved by iterative techniques. A system, in order to permit this type of analysis, must use a mode of operation, methods of input and calculation techniques that minimize the effort required for designer-computer communication so as to maximize effective interaction between the designer and the computer. The time scale is important in process design.

g. Existing Systems Are Not Flexible

Adding new features or modifying some features to the existing systems

is either impossible or a very difficult task. The major limitation to using a state-of-the-art, computer-aided design program to analyze energy conversion processes such as those for coal gasification, oil from oil shale, MHD, etc. is the inability of these systems to handle streams other than conventional vapor-liquid streams. A system applicable to these processes must permit analysis of flowsheets with different types of streams. These include conventional vapor-liquid streams, multi-phase streams containing solids, as well as energy streams and information streams. The system should be sufficiently flexible to allow creation of new stream types, each characterized by a different set of stream variables.

2.5 Thesis Objective

The objective of the thesis was to develop a framework for the development of general purpose chemical process simulators that:

- a. are applicable to all types of chemical processes and
- b. are more adaptable to the design environment.

2.6 Thesis Work

A general framework for modeling of chemical processes has been found and, based upon this framework, a computer system called General Process Engineering System (GPES) has been designed and implemented. GPES allows any group or organization to create its own computer aided design system for engineering of chemical processes. These systems will not have the shortcomings of existing systems that were discussed earlier. These systems could be very simple or very sophisticated depending on the particular needs and applications of the group or organization. Such systems meet the criteria discussed earlier, and therefore provide the design atmosphere for process designers. These systems are not limited to

analyzing process flowsheets having only conventional vapor-liquid streams. They are applicable for analyzing any type of process flowsheet including energy conversion processes such as coal gasification and MHD. Such systems could easily be modified, expanded, and updated. This thesis describes the design and application of GPES. Using GPES several prototype computer aided design systems have been created to demonstrate the application and use of GPES and systems created by GPES.

2.7 General Process Engineering System

GPES is a computer system which enables the rapid production of user-oriented computer-aided design systems for engineering of chemical processes. In using GPES to create a computer aided design system, one has to define different types of process elements (process units, streams, etc.) which may be present in the flowsheets to be analyzed or designed by the users of that system. The subroutines performing the computations for these elements (process modules, etc.) must also be provided. The different types of process elements are defined by providing an information set for each of them. Each such information set is called a template in GPES terminology. For example, a template for a specific type of process unit contains such information as the number of inlet and outlet streams, number of unit parameters, etc. A system created by GPES is called a Template Based System (TBS). The templates for each TBS reside on a set of files called a template data base. Hence, the creation of a TBS consists of creation of a template data base and development of a package of subroutines mainly to represent the mathematical models of process units defined in that template data base. Subroutines are called TBS programs. Each TBS is the responsibility of a system administrator.

A TBS administrator may be assisted by a group of programmers for development of TBS programs. This group is known as TBS programmers.

Once a TBS has been implemented (templates has been defined and computational subroutines provided) a process designer (user) may utilize it to analyze any arbitrary specified configuration of process units which are already defined in the TBS.

The structure of the organization of a team using GPES is shown in Figure 1.1. There are four levels of activity associated with GPES. Each level is the responsibility of a different set of personnel:

1. The GPES administrator who is responsible for:
 - a. Maintenance of the GPES.
 - b. Protecting the system from unauthorized TBS administrators.
 - c. Protecting each TBS from unauthorized users.
2. TBS administrators who are responsible for:
 - a. Implementation and maintenance of template based systems.
 - b. Coordination of TBS programmers.
 - c. Permitting a user to use a TBS.
3. TBS programmers.

They are responsible for development of TBS programs.
4. Users.

They are the ultimate users of the template based systems, the designers of chemical processes. A user may have access to one or more of these systems.

For each of these groups a set of tools and mechanism (programs and languages used to communicate with those programs) has been developed to assist them in performing their responsibilities. GPES consists of these tools.

The use of the system is shown in Figure 1.2. A TBS administrator creates and manipulates his template data base by an interactive program called "update_tdb". A language called Template Definition Language (TDL) has been provided to enable him to easily communicate with the "update_tdb" program to define templates.

TBS programmers write the TBS programs in procedural languages such as PL/1 or Fortran. Development of these programs has been standardized and a package of service routines has been developed to assist them in their efforts.

The backbone of the system is the executive program which provides the means to execute the users' commands. In essence, the executive is a table driven interpreter, the tables being Template Data Bases and GPES files.

The GPES files are part of a mechanism to protect the system from unauthorized TBS administrators and to protect each TBS from unauthorized users. GPES files contain information about each TBS and its authorized users. The GPES administrator performs his administrative duties by creating and updating these files by an interactive program known as the administrative program. The executive program refers to these files to locate the Template Data Base of a specified TBS and to permit a user to use a specified TBS.

A special language has been designed to enable the user to easily communicate with the executive. The language has been named PEL, which stands for Process Engineering Language. PEL consists of a series of commands. Each command is a request for an action to be taken by the system. The language enables a user to create and delete process elements, specify and unspecify parameters and variables of process elements,

calculate (simulate) parts of or whole flowsheets, and print results. PEL allows the choice of engineering units. Many other features are available within PEL.

All information that is related to a particular flowsheet is represented by a network of data structures. This network is created and manipulated by the executive program in response to the user commands. Changes in the flowsheet structure are reflected by changes in the network structure, as the initial concept of the design evolves into a final concept. The network is located in an area of storage known as the working area. The working area automatically expands as the need for more space is recognized. Hence, there is no limitation on the size of the flowsheet being analyzed.

The user can save this network in a file and, later, retrieve it for further analysis. This type of file is known as process file. A user may have any number of process files, each of which may contain any number of processes. The executive program creates and manipulates these files in response to the user's commands. The users may share their process files. This will promote team work on design and analysis of large process flowsheets.

A user may also have any number of another type of files known as component files. A component file may contain the physical properties or constants for estimating these properties for any number of chemical components. The executive program also creates and manipulates these files in response to the user's commands, hence the user need not be explicitly concerned about the creation and maintenance of these files, outside the program. The users may share their component files. A TBS administrator may create such files and make them accessible to all users of that TBS. Such files are known as public component files.

Sharing of process or component files among users may be either unrestricted or restricted depending on the discretion of the owner of the file. Restricted access to a file means that only the information in the file can be read and the contents of the file may not be changed. Unrestricted access, on the other hand, enables one to read or modify the contents of the file.

2.8 The Operating Environment

The system is being implemented in PL/1 on Multics (Multiplexed Information and Computing Services), the time-sharing system by Honeywell [53-59].

The two features of Multics which are of prime importance to the design of GPES are virtual memory and dynamic linking. These are the most significant differences between the Multics programming environment and that of most other contemporary computer programming systems. The latter usually have two sharply distinct environments: a resident file storage system in which programs are created and object code and data are stored, and an execution environment consisting of a central processing unit and a "core image", which contains the instructions and data for the processor. The programs must recognize the existence of and the distinction between the two environments.

In Multics, the line between these two environments is not distinct. Program construction can be simple without sacrificing capability. Programs need be cognizant of only one environment rather than two. This is accomplished by utilizing the concepts of virtual memory and dynamic linking. The Multics' analog of the core image is called an address space. It is different from the usual core image, since it is much larger and is segmented.

A segment is a unit of storage of size up to 256K 36-bit words. User-written programs, object codes, supervisor programs, command procedures, data, etc. are all stored as segments. An address space may have up to 256K different segments. These segments do not reside in the main memory at the same time but are fetched on demand by the operating system. This is done automatically, thus giving the impression that the size of the "core" is 262K million bytes! Hence, the name "virtual memory".

When a program already mapped into the current address space calls another one which is not yet there, a "dynamic linking fault" occurs, the supervisor locates the needed segment and maps it into the current address space. Dynamic linking obviates load modules. Routines are "loaded" into memory only when required.

2.9 Organization of the Thesis

This chapter describes the problems with existing systems and provides the motives for the development of the system. It also provides an overview of the system and its various components. Appendix A is devoted to the review of the state of the art. Chapter 3 presents a framework upon which the design of GPES is based. The design of the Template Data Base is discussed in Chapter 4, while the communication mechanism by which the TBS administrator creates and updates this data base is given in Appendix B. Chapter 5 contains the design of various components of the network of data structures which represent a process flowsheet.

The strategy for development of TBS programs is given in Chapter 6 and the TBS programmer is referred to Appendix C for detailed description of various service routines, which he may want to use in writing his TBS programs. Chapter 7 is devoted to the discussion of the basic principles of the Process Engineering Language (PEL), while Appendix D contains more

detailed description of PEL. The latter serves as the PEL Language Reference Manual for the user. The design of the administrative management aspects of the GPES is given in Chapter 8, which also describes how the GPES administrator should carry out his duties. Chapter 9 is entirely devoted to the design of GPES executive and its various components. The development process of a TBS in general, and the development of three prototype TBS's are described in Chapter 10, which also demonstrates the use and application of these TBS's. Finally, the conclusions of this work, and possible extension to the system and recommended areas for further study are presented in Chapter 11.

CHAPTER 3

A FRAMEWORK FOR THE DEVELOPMENT OF GENERAL PURPOSE PROCESS
SIMULATORS

A "framework" is a way of looking at a problem area. A framework provides us with a set of labels which we can attach to things or concepts. In and of themselves, such labels are of little interest, but if this task is carefully and consistently done, we may find it to be of great help in guiding future actions.

In this chapter we attempt to develop a framework for the development of general purpose process simulators. A general purpose process simulator is a computer tool by which the user can analyze and study chemical processes. A systematic information analysis of a chemical process will provide the framework upon which to develop these systems.

3.1 A Chemical Process

Broadly speaking, the function of every chemical process is to convert materials into more useful products or to convert energy into more useful forms by means of some physico-chemical transformation. Most chemical processes involve an arrangement of individual units of equipment, each of which carries out some step in the overall process. The individual units are interconnected with flow of materials and energy from one unit to the next. Hence, a chemical process may be considered as a collection of process units, and streams transporting chemical species, momenta and energy between individual units.

In other words, a chemical process is a collection of some inter-related entities. These entities fall into one of the following classes: process units, streams, and chemical components. Every piece of information regarding the process can be related to one of these entities.

The method of organizing this information will establish the desired framework, and the following sections describe this methodology.

3.1.1 Chemical Components

Each chemical compound may be characterized by a set of pure-component properties and property estimation constants which we shall refer to as component parameters. The values of these parameters are required for estimation of the thermodynamic and transport properties of streams containing these components.

The values of these parameters may be known or may be estimated either empirically or as a function of externally supplied laboratory data and other component parameters. The estimating procedure, which will be referred to as a component calculating routine, depends on the component type (e.g., hydrocarbon, coal), the laboratory data and the parameters to be estimated. The choice of component parameters will depend upon the type of chemical compound, the extent to which the compound has been studied experimentally, the required accuracy of its properties, and the methods being used to estimate these properties.

In general, components can be classified into different types, each with a particular parameter set and a calculating routine.

A parameter set is defined by the following information:

- 1) The number of parameters.
- 2) The name and the physical dimension of each parameter. Note that all parameters do not have the same physical dimensions. For example, component parameters may have the physical dimensions of temperature or pressure, or be dimensionless.

A component type may or may not require a calculating routine. A component calculating routine may perform different functions or may employ

different mathematical models or estimation techniques. Each of these could be characterized by a number, the level of calculation. A calculating routine may perform different levels of calculation, each with a different set of input/output parameters. In addition to input parameters, a calculating routine may also require other input variables which will be referred to as arguments. The arguments may be for specifying the level of calculation, or they may provide additional data, such as the maximum number of iterations to be used in an iterative search. Therefore, a component calculating routine could be characterized by the following information:

1. The name of the calculating routine.
2. Minimum and maximum number of arguments.
3. The number of levels of calculation.
4. For each level of calculation, the list of input and output parameters. This information is required for controlling the problem of under- or over-specification. In other words, this assures that a value has been assigned to each input parameter and that the values of output parameters have not been fixed by the user before control is passed to the calculating routine.

Hence, a component may be characterized by the following information:

1. A name which identifies the component.
2. The type of the component.
3. The values of component parameters.
4. The value types of component parameters. The value type of a parameter indicates whether the parameter has a value, or how a value has been assigned to the parameter. It indicates whether the parameter is unspecified, assumed, specified or calculated.

This information serves the following purposes:

- a. To indicate whether a value has been assigned to a parameter.
- b. To avoid calculating a parameter which has been fixed (specified) by the user.
- c. To facilitate error checking for under-or over-specification of the input/output parameters.
- d. To indicate which parameters are the output of a calculating routine and, hence, to facilitate the debugging effort.

3.1.2 Streams

In general, a stream may represent the flow of material, momentum, energy and/or information, from one unit in the process (or from the environment) into another unit (or to the environment).

A common type of stream involves the material flow of a single-phased fluid from one unit in the process into another unit. Such a stream might be characterized by the temperature and mass flow rate of each component. A stream that serves only as a heat transfer medium might be characterized simply with two variables: temperature, and the product of mass flow rate and heat capacity. Other types of streams may be defined to describe multi-phase streams, streams containing solids, energy streams, and information streams, each characterized by a set of variables.

In general, each type of stream may be characterized by the following information:

1. What types of components, if any, are allowed to flow in the stream.
2. The number of phases.

3. Two sets of parameters for each phase. One set of parameters represents those attributes of the phase which are independent of the flow of any particular component in the phase. These are called "phase parameters". Temperature, pressure, and total flow rate are examples of phase parameters. Another set of parameters represents those attributes of the phase that are specific to a component in the phase. These component related parameters are referred to as "flow parameters". Mole fraction, particle size distribution, and diffusion coefficients are examples of flow parameters. Each set of parameters, as described earlier in Section 3.1.1, is characterized by: a) number of parameters, and b) name and dimension type of each parameter.
4. The stream calculating routine. Associated with each stream type there may be a calculating routine which performs calculations such as dew point, bubble point, enthalpy, etc. As described earlier in Section 3.1.1, other information associated with such a calculating routine includes: a) minimum and maximum number of arguments, b) number of levels of calculation, and c) for each level of calculation the list of all phase and flow input/output parameters.

One of the phases of the stream may represent the total stream. A phase may correspond to a physical phase of the stream, such as vapor, liquid or solid phases, or it may represent the grouping of certain information about the stream.

All types of streams mentioned earlier can be represented within this framework. For example, a single variable stream can be represented as a stream with no component, one phase, and one phase parameter. The

framework can also represent a multi-phase stream, each phase of which has a number of phase and flow parameters.

Therefore, a stream may be characterized by the following information:

1. A name which identifies the stream.
2. The type of the stream, which provides information on number of phases, component types, etc., as discussed above.
3. Source of the stream which indicates whether it originates from the environment or from a specific outlet port of a unit.
4. Destination of the stream which indicates whether it enters into the environment or into a specific inlet port of a unit.
5. The list of the components present in the stream (if any).
6. For each phase of the stream, the values and value types of all phase parameters.
7. For each phase of the stream, the values and value types of all flow parameters of each component present in the stream.

3.1.3 Units

The most fundamental element in a chemical process flowsheet is the process unit. A process unit may be described in terms of a model which contains a set of mathematical relations between the output stream variables and input stream variables as functions of the values of the design and operating parameters of the unit. A unit type may be characterized by the following information:

1. A set of parameters which represent the design and operating parameters of the unit. As described earlier, each set of parameters is defined by the following information:
 - a. The number of parameters, and,
 - b. The name and physical dimension of each parameter.

2. A description of inlet streams. This includes:
 - a. The number of inlets,
 - b. The name of each inlet port, and
 - c. The type of stream allowed to be connected to each inlet port.
3. A description of outlet streams. This includes:
 - a. The number of outlets,
 - b. The name of each outlet port, and
 - c. The type of stream allowed to be connected to each outlet port.
4. The routine representing the mathematical model of the unit, which is referred to as the unit calculating routine. A unit calculating routine may perform different levels of calculation. This may be due to:
 - a. Different sets of input/output variables. For example a unit calculating routine may offer the following capabilities:
$$\text{outlet streams} = f_1(\text{inlet streams, unit parameters})$$
$$\text{inlet streams} = f_2(\text{outlet streams, unit parameters})$$
$$\text{unit parameters} = f_3(\text{inlet streams, outlet streams}).$$
 - b. Different algorithms for calculation, and
 - c. Different levels of accuracy.The following information is associated with each calculating routine:
 - a. Minimum and maximum number of arguments,
 - b. Number of levels of calculation,
 - c. For each level of calculation the list of the unit input/output parameters,

d. For each level of calculation and for each inlet or outlet port whether a stream should or should not be connected to this port, or whether such a stream is optional. For each stream which is allowable, the list of all phase and flow input/output parameters, and the status of the value of parameters of components that are flowing in the stream, should also be provided.

As described earlier in Section 3.1.1, the above information regarding the calculating routine is required for detection of the problem of under- or over-specification.

A unit, on the other hand, may be characterized by the following information:

1. A name which identifies the unit.
2. The unit type, which associates all the information listed for the unit type to this unit.
3. The values and value types of unit parameters.
4. For each inlet port, the name of the inlet stream (if any).
5. For each outlet port, the name of the outlet stream (if any).

3.2 Other Elements of Interest

A General Purpose Process Simulator, if it is to be effective as a design tool, must also provide the user with the capability of analyzing experimental data and performing side calculations. Therefore, in design of these simulators, in addition to basic elements of a chemical process (units, streams, and components), two other classes of entities, namely functions and variables, should also be considered.

3.2.1 Functions

The purpose of functions is to enhance the user's calculating ability and to provide him the capability of analyzing experimental data.

The user should be able to perform standard mathematical operations such as SQRT, LOG, SIN, COS, ETC. These functions could be built in (built-in functions) to the system and the user would be free to use them. The user should also be able to define his own analytical functions to improve his problem solving capability. These functions are called user-defined functions. These two classes of functions do not provide the user with any data analysis, capability, or the ability to evaluate non-analytical functions. To overcome these shortcomings, we introduce another class of functions called pre-defined functions. A pre-defined function may be thought of as an entity with a set of parameters and a calculating routine. The latter may calculate the function parameters as a result of the analysis of some raw data. In addition, another routine may be associated with a pre-defined function to evaluate the function. This routine, which is referred to as a pre-defined function evaluating routine may evaluate any analytical, non-analytical, or numerical function. Each pre-defined function type may be characterized by the following information:

1. A set of parameters presenting the coefficients of the function. As described earlier, this includes: a) number of parameters and b) name and physical dimension of each parameter.
2. The evaluating routine and its required number of arguments.
3. The calculating routine and its associated information: a) minimum and maximum number of arguments, b) number of levels of calculation, and c) input/output parameters for each level of calculation.

Therefore, a pre-defined function may be characterized by the following information:

1. The name which identifies the function.
2. The type which associates the above listed information for the pre-defined function type to this function.
3. The value and value type of each function parameter.

3.2.2 Variables

A variable is an entity which represents a value. A variable may be classified to the following groups:

1. Simple or user-supplied variables. Such a variable may be characterized by a name and a value.
2. Qualified variables. A qualified variable is a variable referring to a parameter of a unit, stream, component, or pre-defined function.
3. Built-in variables or built-in constants. These are various constants commonly used by chemical engineers, such as the gas constant (R), Faraday's constant (F), Avogadro's constant (N), etc. The value of these constants should be known by the system.

3.3 Units of Measurement

Every variable of a process flowsheet that can be represented with numerical data is called a "parameter". There are five kinds of parameters:

1. Unit Parameters
2. Stream Phase Parameters
3. Stream Flow Parameters
4. Component Parameters
5. Function Parameters

The number of stages in a distillation column would normally be defined as a unit parameter of the column. The temperature of a stream would normally be termed a stream phase parameter. The molar flow rate of a component in a stream or its mole fraction would usually be a flow parameter. The critical temperature of a component would be a component parameter. Regression coefficients of a function can be defined as function parameters.

All the parameters do not have the same physical dimensions. The number of stages in a distillation column is dimensionless while the critical temperature of a component has a physical dimension of "temperature". A mechanism should be available to enable a user to provide parameters in any units of measurement.

Such a mechanism requires the following information for each physical dimension:

1. Standard units of measurement.
2. The number of other allowable options.
3. For each option, the units of measurement and some conversion factors to enable the conversion of the optional units into the standard units.

3.4 Calculating Routines

Calculating routines are the heart of any process simulator. As mentioned earlier, there may exist a calculating routine for every type of unit, stream, component and function.

The unit calculating routine is usually referred to as a "process module" or "building block" in process simulation literature. The function of a unit calculating routine is to simulate or design process units. The concept of stream and component calculating routines is not often used in other simulators.

A stream calculating routine may calculate some stream parameters as a function of other parameters. Examples are calculation of dew point, bubble point, enthalpy, viscosity, and phase equilibrium.

A component calculating routine may calculate a component's parameter as a function of other parameters and experimental data. A function calculating routine may perform regression analysis and other data analysis functions.

As noted earlier, a calculating routine may perform different levels of calculation. This may be due to:

1. Different sets of input/output variables.
2. Different algorithms for calculation.
3. Different mathematical models, or different levels of accuracy.

For each level of calculation there is a particular set of parameters as the input to the routine, and another particular set of parameters as the output from the routine. Each input parameter should have a value and the values of output parameters should not have been fixed by the user. Therefore, for each level of calculation the input and output parameters should be checked for under- or over-specification. As noted earlier, a calculating routine, in addition to input parameters, may also require other input variables, usually computational variables which will be referred to as arguments. The computational variables are data such as the maximum number of iterations to be used if the routine is performing an iterative search. These computational variables may also be considered as a part of the input parameters. The arguments may also be used for specifying the level of calculation, or specifying other options that a calculating routine may provide.

3.4.1 Route Selection

A routing problem arises when a calculating routine calls upon another routine, which in turn calls upon other routines, with choices existing at each level. Seider, et. al. [151,166,32] discuss the problem of routing and propose a method for implementing the routing capability (the ability to specify the sequence of computations) in process simulators. The nature of routing problems is illustrated through an example shown in Figure 3.1. Consider the calculating routine "C1" which offers the choices of two calculational methods: "C11" and "C12". Each calculational method will call upon various physical property estimation routines (e.g., P1, P2), which in turn call upon other physical property estimation routines (e.g. P3, P4, P5). Consider also that each physical property estimation routine offers the choice of different methods of estimation (e.g., P11, P12, P21, P22, etc.). Seider's method allows the user to completely specify the route of computation. For example, the user may specify the route as shown by thick lines in Figure 3.1. Note that different methods of estimation are used to estimate property P3. Method P31 is used for property P2, while method P32 is used for property P1. A simpler approach could be used which results in a more limited capability of route selection. That is to allow the user only to specify the method to be used for each property estimation, independent of the route of computation. With this approach the method of estimation of property P3 may be either P31 or P32 for all computations associated with calculating routine C1. Of course, the user should always be able to alter his choices at any time, or to use default options. In this approach there should be a global data base which contains the methods currently being used for each property estimation.

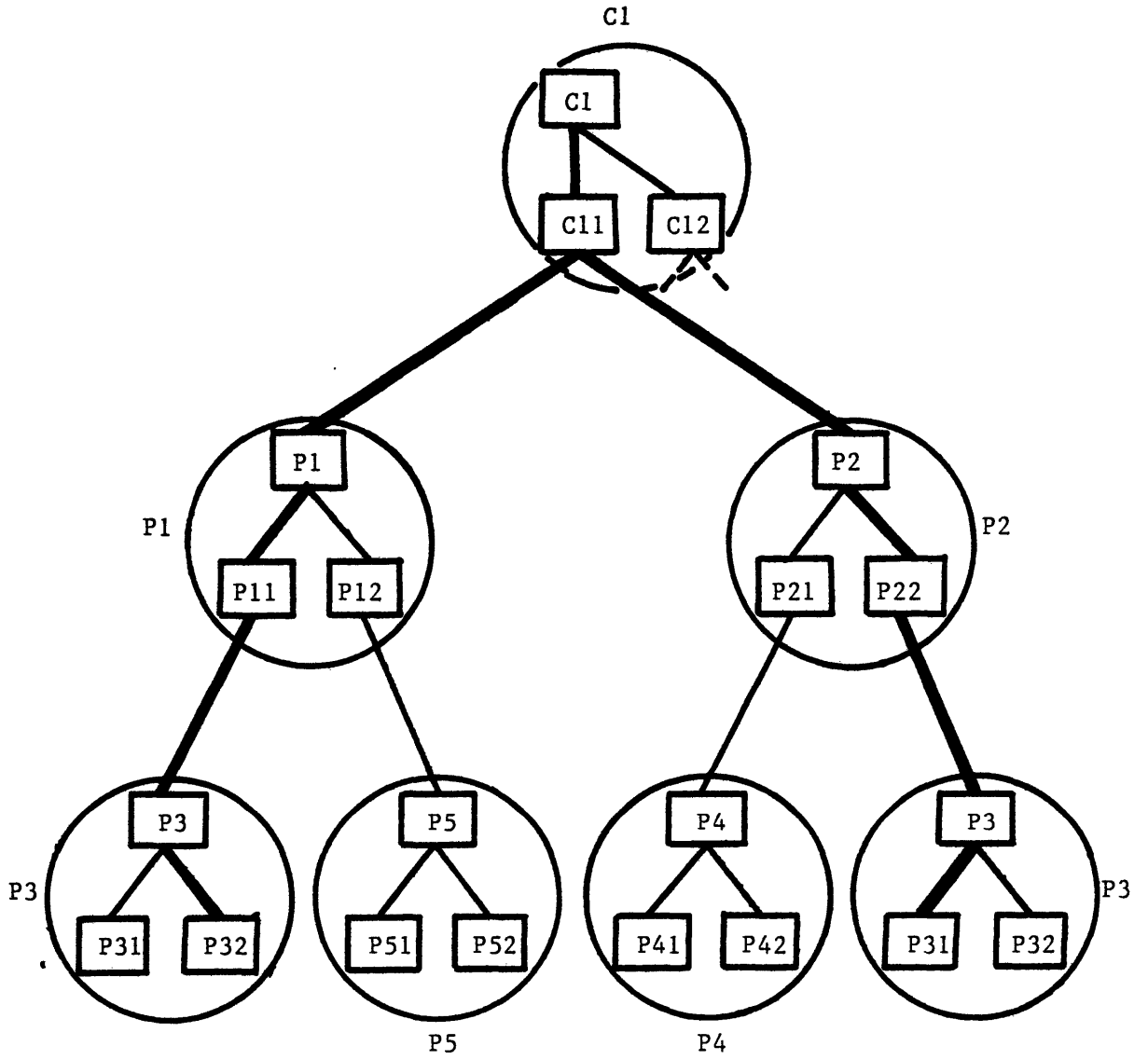


FIGURE 3.1 AN EXAMPLE OF ROUTINING PROBLEM

This data base should be updated as the user alters his choices. The following information would be required to manage this mechanism:

1. number of properties to be estimated.
2. name of each property.
3. number of options available for each property estimation.
4. the default option for each property estimation.

The argument list mentioned earlier is also another mechanism to supply the level of calculation and other computational variables to the calculating routine.

3.5 Template Based System

The design of the GPES is based upon the framework established in this chapter. GPES differs also from most computer simulation systems in its representation of the process flowsheet. Process elements such as units, streams, components, functions (pre-defined, user-defined), and variables (user-supplied) are represented by data structures connected to form a network. This network represents the process flowsheet and is manipulated by the system in response to the user commands as the initial concept of the design evolves into a final concept. This will permit the user to retain an active model of the problem being solved. The resulting model may be saved indefinitely for later consideration. GPES employs generalized data structures to represent process elements. However, before the system can utilize them to model a process configuration, it requires certain information about the types of units, streams, components, etc. This information about each type of each process element (unit, stream, etc.) is stored in a data structure called a "Template" in GPES terminology. As a drawing template enables one to draw any number of

diagrams represented in that template, a template data structure enables the system to create other data structures representing the process elements defined by that template. For example, a template of stream type x enables the system to create data structures, each representing a stream of type x. Templates also contain other information, such as information regarding the calculating routines, error checking to be done before control is passed to the calculating routine, etc. The collection of these templates and associated calculating routines (TBS programs) is called a template based system (TBS). The templates are stored in a set of files called the template data base. The information in this data base is controlled by the TBS administrator. To enable the the TBS administrator to insert, update, or delete the information in the data base, an interactive program called "update_tdb" (update template data base) has been developed along with a set of commands to control the operations of this subsystem. The set of commands is called Template Definition Language (TDL). The design of the template data base and other issues related to the template data base is given in the next chapter.

CHAPTER 4THE TEMPLATE DATA BASE

As described earlier, GPES creates a network of data structures, to represent a chemical process. To create this network, however, it requires certain information about the types of units, streams, components, etc. This information is stored in the form of "templates" in a data base called the "Template Data Base". The information in this data base is controlled by the TBS system administrator. To enable him or her to insert, update or delete the information in the data base, an interactive program called "update_tdb" has been developed along with a set of commands to control program operations. The set of commands is called Template Definition Language (TDL).

4.1 The Information Content of the Template Data Base

The information in the data base is stored in the form of PL/1-type data structures. All information is related to one of the following:

1. The Unit Types
2. The Stream Types
3. The Component Types
4. The Function Types
5. The Property Estimation Methods
6. The Physical Quantity Dimensions (units) and Unit Conversions
7. System Control Information
8. Text Information

The following terms and structures need to be explained before describing the above information categories.

A. Parameter Template Structure

As described earlier in section 3.3, every variable of a process flowsheet that can be represented by a numerical value is called a parameter in GPES terminology. A parameter template structure as shown in Figure 4.1 defines a parameter set. Each parameter is represented by a name and a "Dimension Type". The name is the identifier used by users to refer to that parameter. The "Dimension Type" is a number which is defined in the dimension table, whose detailed description follows later. For each dimension type the dimension table contains the standard units of measurement (dimension) and other allowable units with their conversion factors. Associating a dimension type for each parameter and providing the dimension table enables the user to provide his input in any allowable units of measurement. The system will automatically convert the input to the standard units of measurement and store the result in an appropriate location. Hence, the calculating routines always receive the input parameters as standard units.

B. Calculating Routines

Information regarding the calculating routines is a part of information stored in a template. Before describing this information the interaction between the GPES executive and the calculating routines will be discussed. Calculating routines are invoked in response to the user's "calculate" commands. For example, suppose units A and B are both of type heat_exchange and the calculating routine of unit type heat_exchange is called heatx. Routine heatx will be called twice in response to the following command: calculate unit (A, B (1, 12, 4));.

In effect the heatx routine is invoked as follows: call heatx (punit, parg, switch, error_switch); where punit is a pointer to the unit

PARAMETER TEMPLATE STRUCTURE

NUMBER OF PARAMETERS = N		
1	NAME	DIMENSION TYPE
2		
N		

FIGURE 4.1 THE PARAMETER TEMPLATE STRUCTURE

structure. For the first call on routine heatx, punit points to the unit "A" data structure, and on the second call it points to the unit B data structure (unit data structures will be described in Chapter 5). The variable parg is a pointer to a list of arguments in a data structure such as shown in Figure 6.3. During the first call parg is null, for the second call it points to an argument data structure containing the three supplied arguments 1, 12, and 4. The argument switch indicates whether convergence has been achieved, and is only used for special types of calculating routines known as convergence routines. The argument error_switch indicates whether a serious error has been detected in the routine. The argument list, such as the one in the above command, may be used to specify the calculation route or to provide additional input to the routine. In any case, the interpretation of these arguments should be clear both to the user and the calculating routine.

The first argument is always interpreted as the level of calculation. If it is not provided the default level, i.e. level one, will be assumed. As discussed in section 3.4, a calculating routine may perform different levels of calculation. This may be due to different sets of input/output variables, different algorithms for calculation, different mathematical models, or different levels of accuracy.

The following information is stored in a template for a calculating routine:

Procedure: Name of the calculating routine. If no routine has been implemented or no routine is needed a ";" is placed in this field and no other information about the calculating routine will be required.

Reference: A short description of the calculating routine or the template; or a reference to a document describing the calculating routine. It could also contain information such as author, date, etc.

Minimum Number of Arguments:

This is the minimum number of arguments that the routine requires. It should be 0 or more. It implies that the user should provide at least this number of arguments in a calculate command, invoking the routine.

Maximum Number of Arguments:

This is the maximum number of arguments that can be passed to the routine. A negative number in this field indicates that the maximum number of arguments is unlimited. This number must be at least 1; in effect this always gives the user the option of specifying the level of calculation (first argument).

Number of Levels of Calculation:

This is the number of levels of calculation that the routine provides. It should be one or more. Therefore, the first argument provided by the user should not be greater than this number. If no argument is provided the level of calculation would be assumed to be one.

The template provides also information regarding the error checking to be done by the system for each level of calculation before control is passed to the calculating routine. This checking is primarily for detecting under- or over-specification for the calculating routine.

C. Value Status of Parameters

As will be described in Chapter 5, the numerical values of parameters are stored in parameter data structures such as the one shown in Figure 5.3. With every parameter value is associated a value type. This is entirely different from the dimension type mentioned previously and is associated with a parameter value, not the parameter itself. The value type of the parameter indicates how a value has been assigned to the parameter. The value type can be one of the following:

1. Unspecified (value type = 0)
2. Assumed (value type = 1)
3. Specified (value type = 2)
4. Calculated (value type = 3)

When control is given to the calculating routine, for each level of calculation it expects certain parameters to have one of certain types and not others. The Value Status Template structure of Figure 4.2 is a way of telling the GPES executive to check that the parameters have values only of certain types. Each entry of this data base has four single "bit" flags. If, for instance, flags 1 and 3 are turned on for a parameter value, GPES executive will check to see that the associated type is either "unspecified" or "specified" but neither "assumed" nor "calculated".

The four flags, when concatenated, form a 4-bit binary number ranging from 0 to 15. This is called the "Value Status Code" and its only significance is the convenience it offers the TBS administrator when he is entering this information at the terminal.

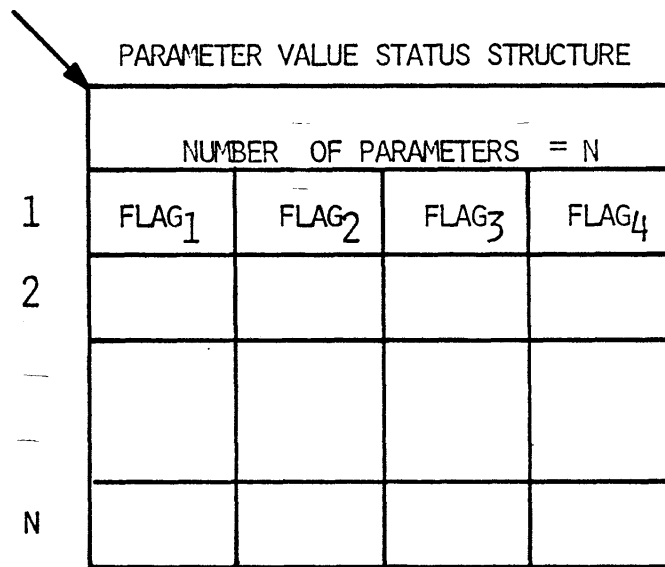


FIGURE 4.2 THE PARAMETER VALUE STATUS STRUCTURE

Most of the parameters occurring in a computation are either:

1. Input variables,
2. Output variables, or
3. Initial assumption variables.

The input variables should not be "unspecified". The output variables should not be "specified" and the initial assumption variables should neither be "specified" nor "unspecified". Thus, the most commonly occurring value status codes are:

	<u>Value Status Code</u>	<u>4-bit Representation</u>
1. Input variables	7	0111
2. Output variables	13	1101
3. Initial Assumption variables	5	0101

As is evident, the value status code of 0 is not acceptable. Table 4.1 gives the meanings of all the value status codes (this table is the output of TDL command: "print vsctable" or PEL command: "printt vsctable ;").

The description of various template structures follows:

4.1.1 The Dimension Table

As described earlier, there is a dimension type associated with each parameter in the parameter template structure. The dimension type indicates the physical dimension of the parameter and consequently indicates the allowable units of measurement. Dimension types have been represented by integer numbers: 0,1,2,... This choice has primarily been based on efficiency of processing (searching) and saving storage. The dimension type of zero is used for dimensionless parameters. Each entry of the data structure shown in Figure 4.3 corresponds to a dimension type, and consists of the following:

TABLE 4.1

STATUS CODE	VALUE STATUS CODES TABLE			CALCULATED
	UNSPECIFIED	ALLOWABLE PARAMETER TYPE ASSUMED	SPECIFIED	
1(0001)				x
2(0010)			x	
3(0011)			x	x
4(0100)		x		
5(0101)		x		x
6(0110)		x	x	
7(0111)		x	x	x
8(1000)	x			
9(1001)	x			x
10(1010)	x		x	
11(1011)	x		x	x
12(1100)	x	x		
13(1101)	x	x		x
14(1110)	x	x	x	
15(1111)	x	x	x	x

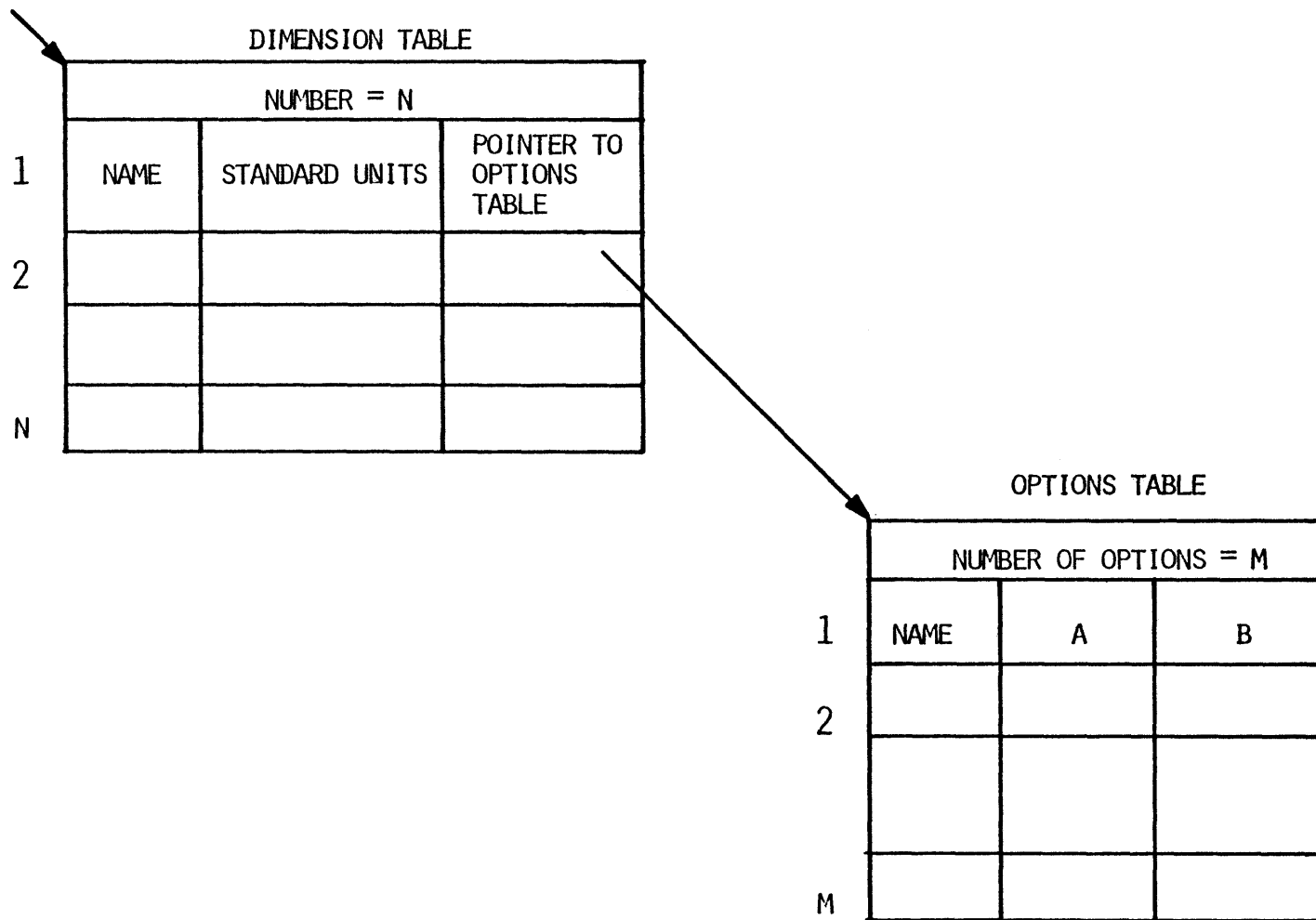


FIGURE 4.3 THE DIMENSION TABLE STRUCTURE

- a) name of the physical dimension (e.g., temperature, pressure, etc.)
- b) standard units of measurement (e.g., DEGF)
- c) a pointer to a data structure which contains other options with their conversion factors.

There are two constants for converting a value in optional units to the standard units. A linear relation is assumed such as

$$y = A + B x$$

where: y = the value in standard units,
 x = the value in other allowable units, and
 A and B are conversion factors.

When the user provides his input in optional units of measurement the executive will automatically convert the input to the standard units of measurement by using the above conversion formula. An example of a dimension table is shown in Figure 4.4.

4.1.2 Stream Templates

A stream template contains a set of information defining a stream type. All stream templates are placed in a directory to facilitate the search for the template of a stream type. The directory is called "Stream Template Directory" and is shown in Figure 4.5. Each entry of the directory represents a stream template. The first entry represents the template of the default stream type. Each entry contains the following information:

- a) stream type
- b) the name of the calculating routine or a ";"
- c) reference information
- d) minimum number of arguments for the calculating routine
- e) maximum number of arguments for the calculating routine

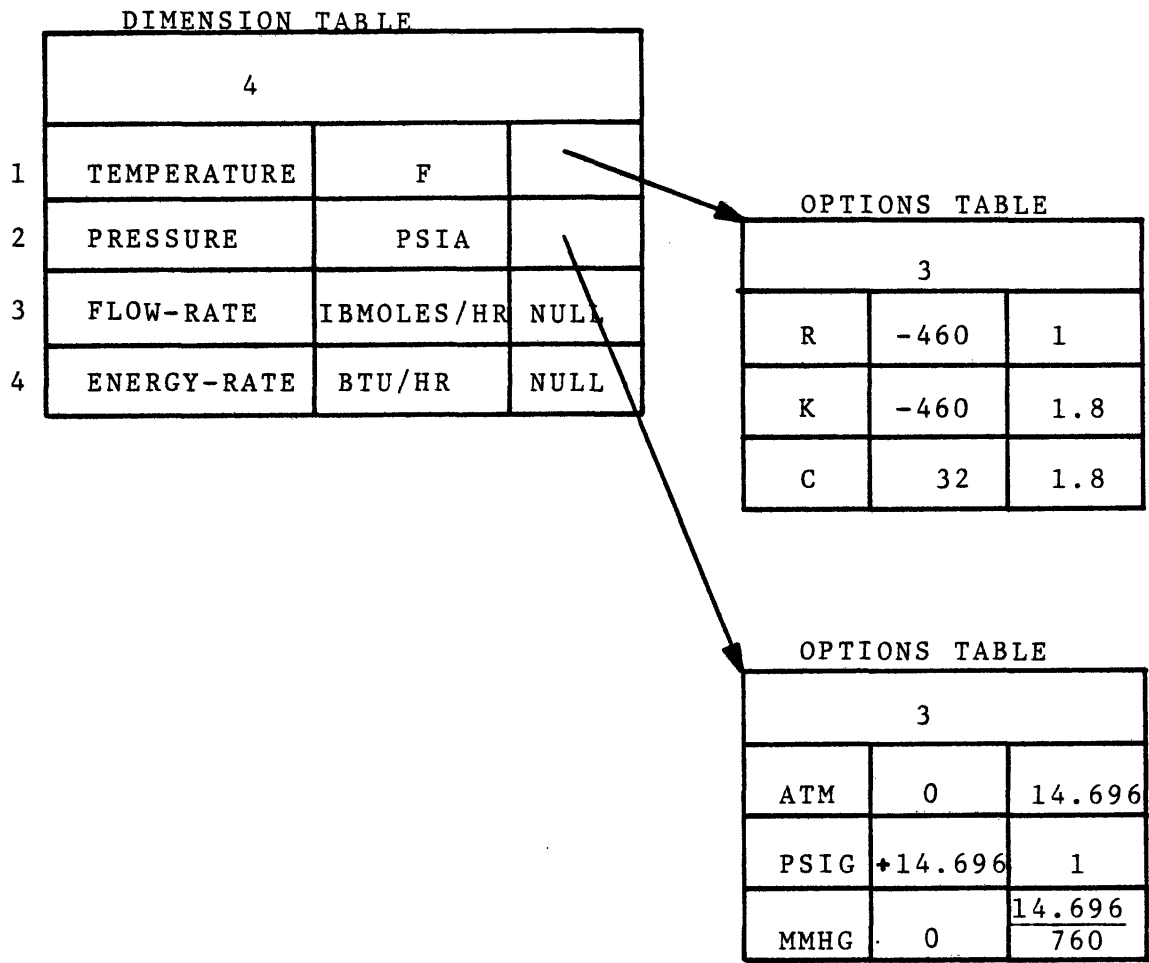


FIGURE 4.4 AN EXAMPLE OF A DIMENSION TABLE

STREAM TEMPLATE DIRECTORY

NUMBER OF STREAM TYPES = n								
1	TYPE	PROCEDURE	REFERENCE	MINIMUM NUMBER OF ARGUMENTS	MAXIMUM NUMBER OF ARGUMENTS	COMPONENT TYPE	POINTER TO PHASE TEMPLATE	POINTER TO LEVEL SUBSTRUCTURE
n								

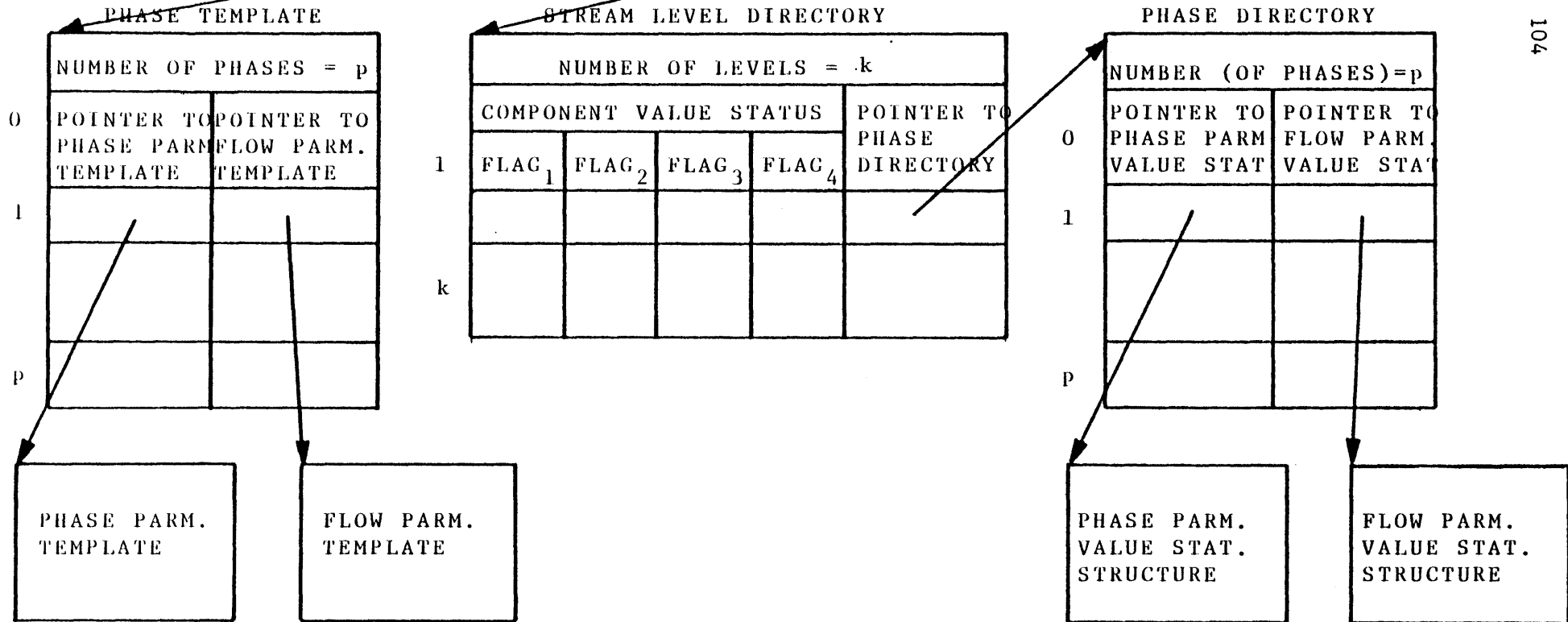


FIGURE 4.5 THE STREAM TEMPLATE STRUCTURES

- f) type of components (chemical species) allowed to flow in the stream. Using the symbol "none" in this field indicates that no component is allowed to flow in the stream, and the symbol "all" indicates that any type component is allowed to flow in the stream.
- g) a pointer to a data structure which describes the different phases of the stream. The data structure is called "phase template". Phase zero usually represents the total stream. A stream may have any number of phases. The "phase template" contains the number of phases and an entry for each phase. Each entry contains two pointers; one pointer points to a parameter template structure describing the phase parameters and another pointer points to a template structure describing the flow parameters. The phase parameter template structure contains the number of phase parameters and the name and dimension type of each parameter. The flow parameter template structure contains the number of flow parameters and the name and dimension type of each parameter.
- h) a pointer to a data structure which contains the number of levels of calculation and for each level the error checking to be performed by the system before control is passed to the calculating routine. For each level of calculation there is an entry in this data structure. Each entry contains the following:
- i) four flags indicating the value status codes of all parameters of all components flowing in the stream (if any).

- ii) a pointer to the data structure "phase directory", which contains the value status codes of each phase and flow parameters of each phase of the stream.

An example of a stream template directory which contains only one stream type is shown in Figure 4.6. The stream has one phase (phase number 0), 5 phase parameters and 1 flow parameter. Phase parameters are:

- tf - Total molar flow rate
- t - Temperature
- p - Pressure
- h - Enthalpy flow rate, and
- vf - Vapor fraction (molar).

The flow parameter is molar flow rate (f).

4.1.3 Unit Templates

A unit template contains the information defining a process unit type. All unit templates are placed in a directory to facilitate search procedures. The directory is called the "Unit Template Directory" and is shown in Figure 4.7. Each entry of this directory represents a unit template. The first entry represents the default unit type. Each entry contains the following:

- a) unit type (e.g., heat_exchanger)
- b) name of the calculating routine or a ";"
- c) reference information
- d) minimum number of arguments
- e) maximum number of arguments
- f) a pointer to a parameter template structure describing the unit parameters. The unit parameter template structure contains the number of unit parameters, name and dimension type of each parameter.

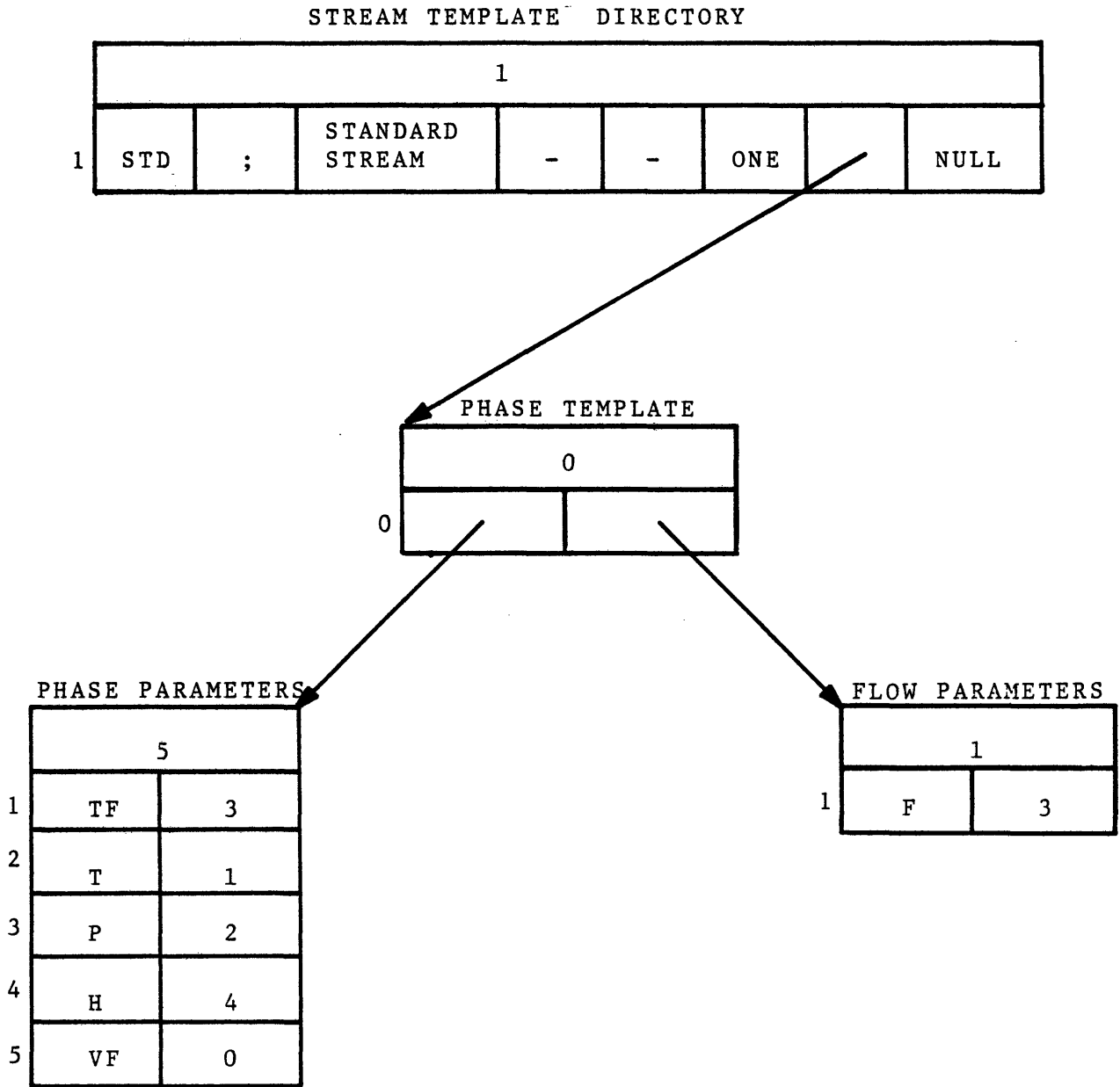


FIGURE 4.6 AN EXAMPLE OF A STREAM TEMPLATE

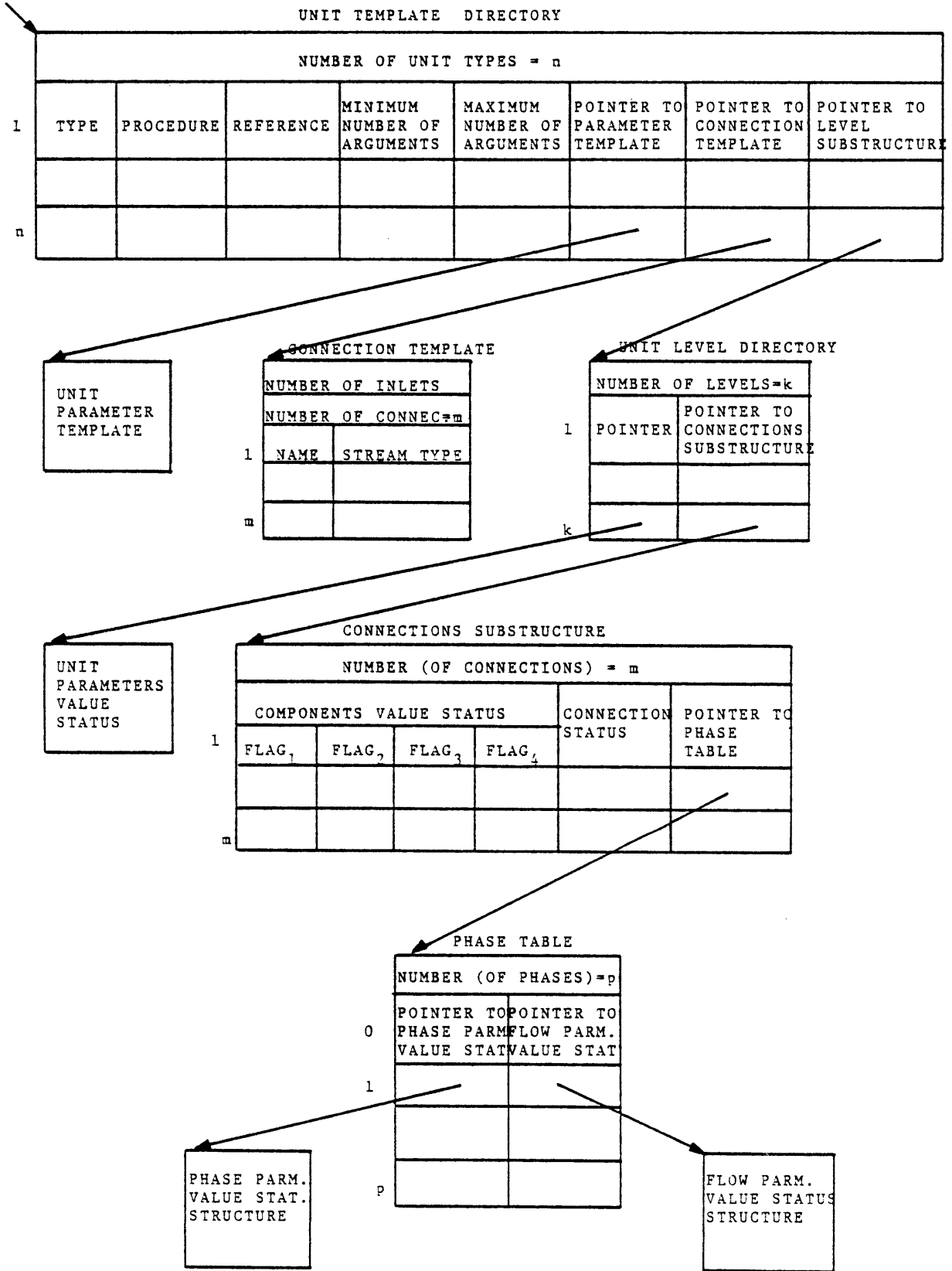


FIGURE 4.7 THE UNIT TEMPLATE STRUCTURES

- g) a pointer to the connections template structure describing the streams flowing into and out of the unit. The connections template structure contains: the number of inlets (0 or more), the number of connections (inlets and outlets), the name of each connection, and the type of streams allowed to be connected to each connection. The symbol "all" in the latter field indicates that any type stream could be connected.
- h) a pointer to a data structure which contains the number of levels of calculation and for each level of calculation specifies the error checking to be performed by the system, before control is passed to the calculating routine. The first entry of this data structure is associated with the level one calculation and the second entry is associated with the level two calculation, and so on. Each entry contains two pointers: one points to a parameter value status structure describing the status of each unit parameter's value; and another one points to the connections substructure. The connection substructure describes the checking procedures for the unit connections. The connection substructure has as many entries as the unit has connections. For each entry of this structure there is a connection status. It is a number indicating whether a stream should or should not be connected to that connection. A connection status of 1 indicates that such a stream is required. A connection status of 2 indicates that such a stream is not permitted. A connection status of 3 indicates that such a stream is optional. If a specified stream type can be connected to a connection (connection

status is one, or three and stream type of the connection is not "all") then the entry contains the following additional information:

- i) value status of all parameters of all components (if any) flowing in the connected stream.
- ii) a pointer to a data structure describing the value status of each phase or flow parameter of each phase of the stream.

An example of a unit template is shown in Figure 4.8. The unit is a splitter. It has 1 parameter, 1 inlet, and 2 outlets. The unit parameter is the splitting ratio (R) and the inlet and outlet streams are of type "STD" as defined earlier in Figure 4.6.

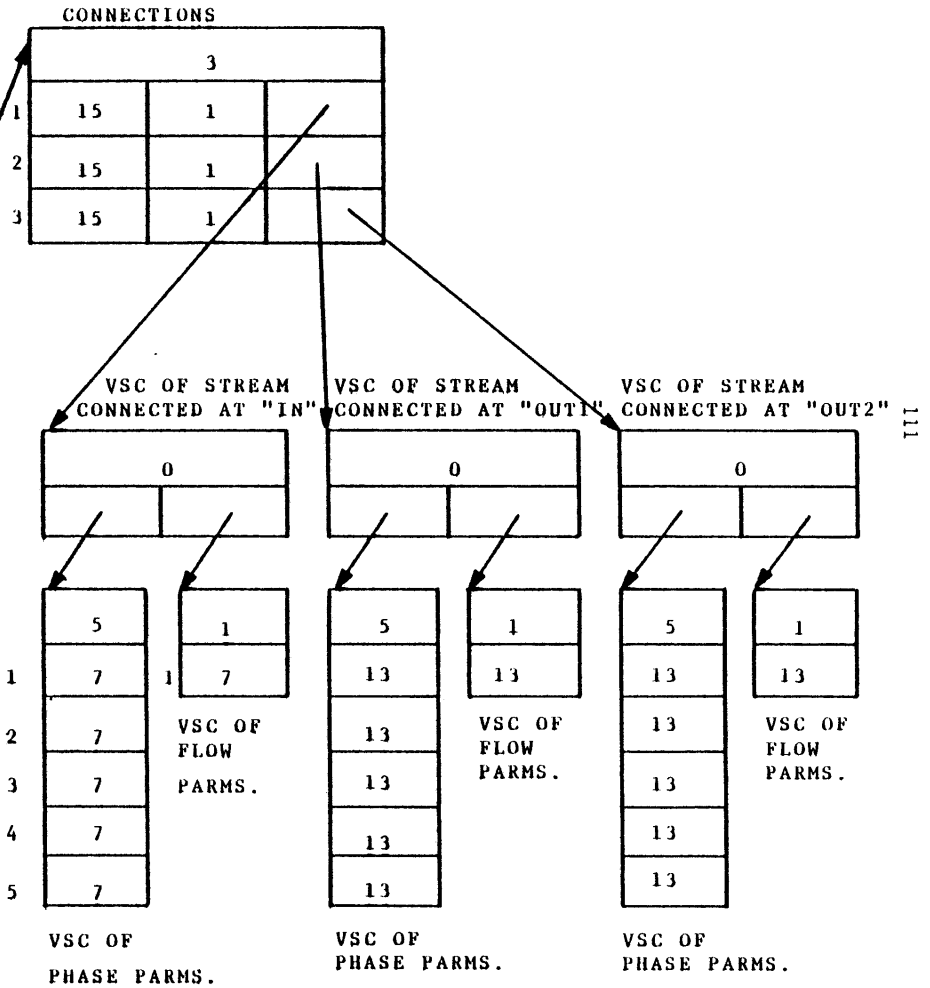
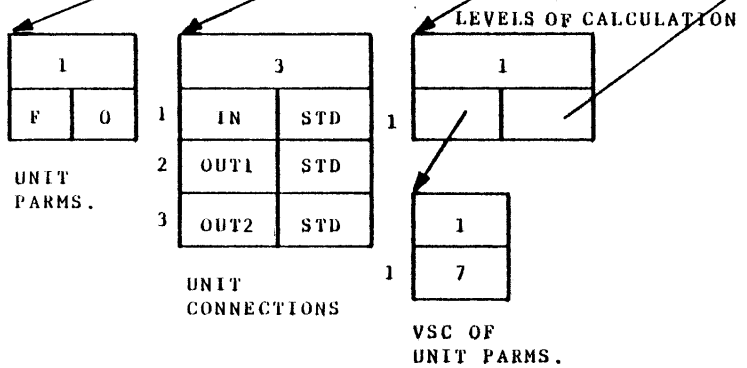
4.1.4 Component Templates

A component template contains information defining a component type. All component templates are placed in a directory to facilitate the search for the template of a given component type. The directory is called "Component Template Directory" and is shown in Figure 4.9. Each entry of the directory represents a component template. The first entry represents the template of the default component type. Each entry contains the following information:

- a) component type.
- b) the name of the calculating routine or a ";".
- c) reference information.
- d) minimum number of arguments for the calculating routine.
- e) maximum number of arguments for the calculation routine.

UNIT TEMPLATE DIRECTORY

SPLITTER	SPLIT		0	1			



NOTATIONS

VSC - VALUE STATUS CODE
 PARMS. - PARAMETERS

VSC	DESCRIPTION
7	INPUT VARIABLE
13	OUTPUT VARIABLE
15	NEITHER INPUT NOR OUTPUT

FIGURE 4.8 AN EXAMPLE OF A UNIT TEMPLATE

COMPONENT TEMPLATE DIRECTORY

NUMBER OF COMPONENT TYPES = n

	TYPE	PROCEDURE	REFERENCE	MINIMUM NUMBER OF ARGUMENTS	MAXIMUM NUMBER OF ARGUMENTS	POINTER TO PARAMETER TEMPLATE	POINTER TO LEVEL SUBSTRUCTURE
1							
2							
...							
n							

COMPONENT
PARAMETERS
TEMPLATE

COMPONENT
PARAMETERS
VALUE STATUS
STRUCTURE

COMPONENT LEVEL DIRECTORY

	NUMBER OF LEVELS=k
1	POINTER TO PARAMETERS VALUE STATUS STRUCTURE
...	
k	

FIGURE 4.9 THE COMPONENT TEMPLATE STRUCTURES

- f) a pointer to a parameters structure describing the component parameters. The parameters structure contains the number of component parameters, and the name and dimension type of each parameter.
- g) a pointer to a data structure containing the number of levels of calculation and for each level of calculation information regarding the error checking to be carried out by the system, before control is passed to the calculating routine. The first entry of this data structure is associated with the level one of calculation and the second entry is associated with the level two and so on. Each entry points to a parameters value status structure describing the status of each component parameter's value.

An example of a component template directory having only one component type is shown in Figure 4.10. The component has 8 parameters. Molecular Weight (MW), Normal Boiling Point (NBP), Critical Temperature (TC), Critical Pressure (PC), Critical Compressibility Factor (ZC), and Antoine Vapor Pressure Constants (A1,A2,A3).

4.1.5 Function Templates

A function template contains information defining a function type. All function templates are placed in a directory to facilitate the search for the template of a given function type. The directory is called "Function Template Directory" and is shown in Figure 4.11. The first entry represents the default function type. Each entry contains the following information:

COMPONENT TEMPLATE DIRECTORY

1						
STD	;	STANDARD TYPE COMPONENT	-	-		NULL

	8	
1	MW	0
2	NBP	1
3	TC	1
4	PC	2
5	ZC	0
6	A1	0
7	A2	0
8	A3	0

FIGURE 4.10 AN EXAMPLE OF A COMPONENT TEMPLATE

FUNCTION TEMPLATE DIRECTORY

NUMBER OF FUNCTION TYPES = n									
1	TYPE	PROCEDURE TO CALCULATE THE FUNCTION	REFERENCE	MINIMUM NUMBER OF ARGUMENTS	MAXIMUM NUMBER OF ARGUMENTS	PROCEDURE TO EVALUATE THE FUNCTION.	NUMBER OF ARGUMENTS FOR EVALUATING THE FUNCTION	POINTER TO PARAMETER TEMPLATE	POINTER TO LEVEL SUBSTRUC.
2									
n									

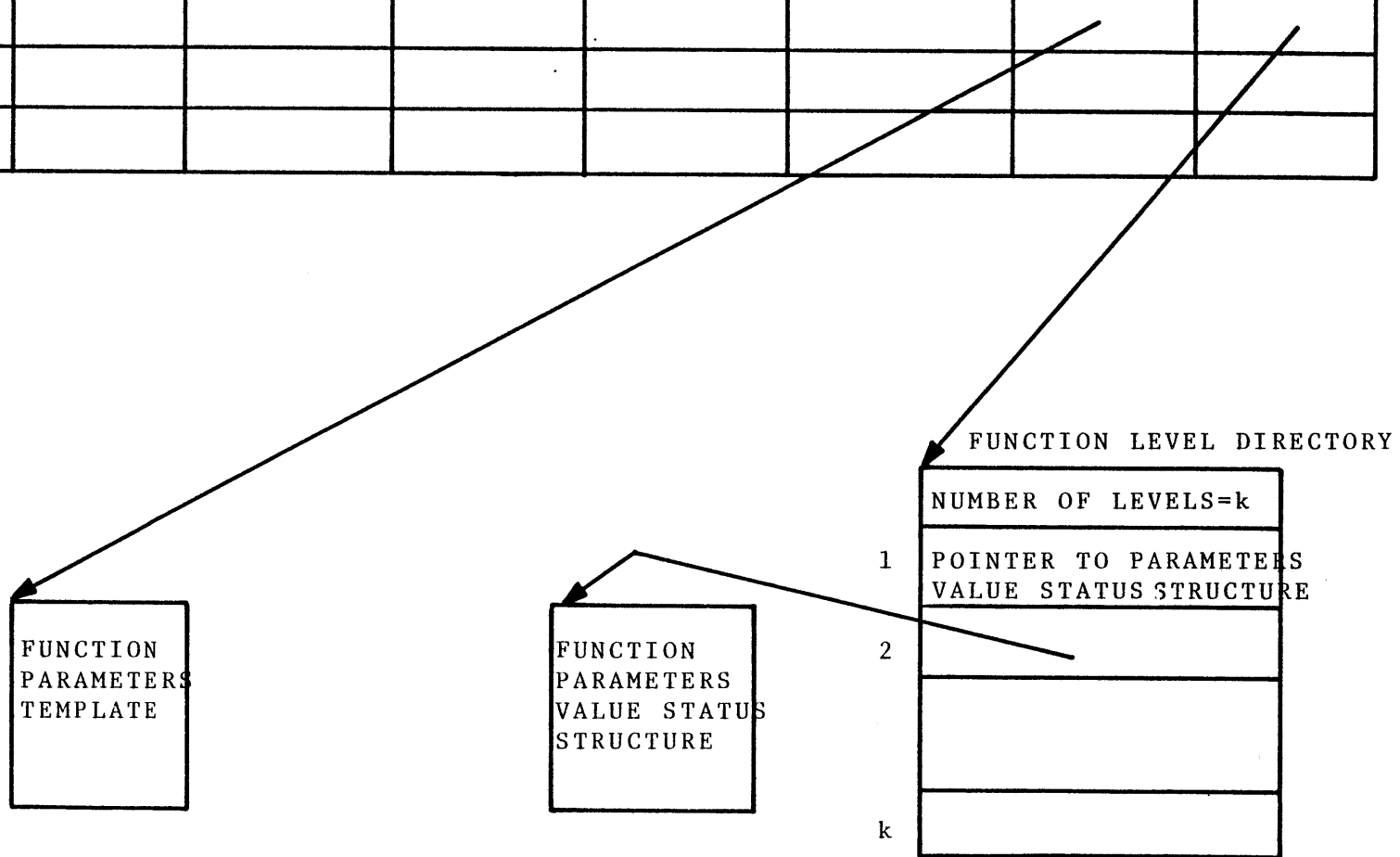


FIGURE 4.11 THE FUNCTION TEMPLATE STRUCTURES

- a) function type,
- b) name of the function calculating routine (e.g., Regression Analysis Program), or a ";"
- c) reference information,
- d) minimum number of arguments for the calculating routine,
- e) maximum number of arguments for the calculating routine,
- f) name of the procedure to evaluate the function value or a ";". This is the routine that will be called when a pre-defined function appears in an arithmetic expression,
- g) the number of arguments for the above routine,
- h) a pointer to a parameter template structure describing the function parameters. The function parameter template structure contains the number of function parameters, the name and dimension type of each parameter.
- i) a pointer to a data structure containing the number of levels of calculation and for each level the specific checking procedures to be carried out by the system, before control is passed to the calculating routine. The first entry of this data structure is associated with the level one of calculation, and the second entry is associated with the level two and so on. Each entry points to a parameters value status structure describing the status of each function parameter's value.

An example of a function template directory having only one function type is given in Figure 4.12. The function is in the form: $y = A+Bx$. The function parameters (coefficients) A and B may be calculated by routine L1CALC. Routine L1EVAL evaluates the function for a given argument, x.

4.1.6 Table of Property Estimation Methods

A calculating routine may call upon other routines, especially for physical property computations. Usually there is more than one method for estimating a physical property. A mechanism is provided to enable a user to specify the particular methods to be used for estimation of physical properties. The Table of Property Estimation Methods as shown in Figure 4.13 is the essential part of this mechanism. Each entry of the table corresponds to a property and contains the following:

- a) the property name as known by the users.
- b) the number of options available for estimating that property.
- c) the default option. This is the option that will be in effect until the user chooses another option for estimation of that property.

The entry number of the property in the table is called the property number or property type. This is the number used by the physical property estimation routine to retrieve its specified method of calculation. An example of a property estimation methods table is shown in Figure 4.14.

4.1.7 Control Information

A set of miscellaneous information is associated with each TBS as shown in Figure 4.15. It consists of:

- a) the name of the Template Based System.
- b) the serial number of the TBS, identifying the

FUNCTION TEMPLATE DIRECTORY

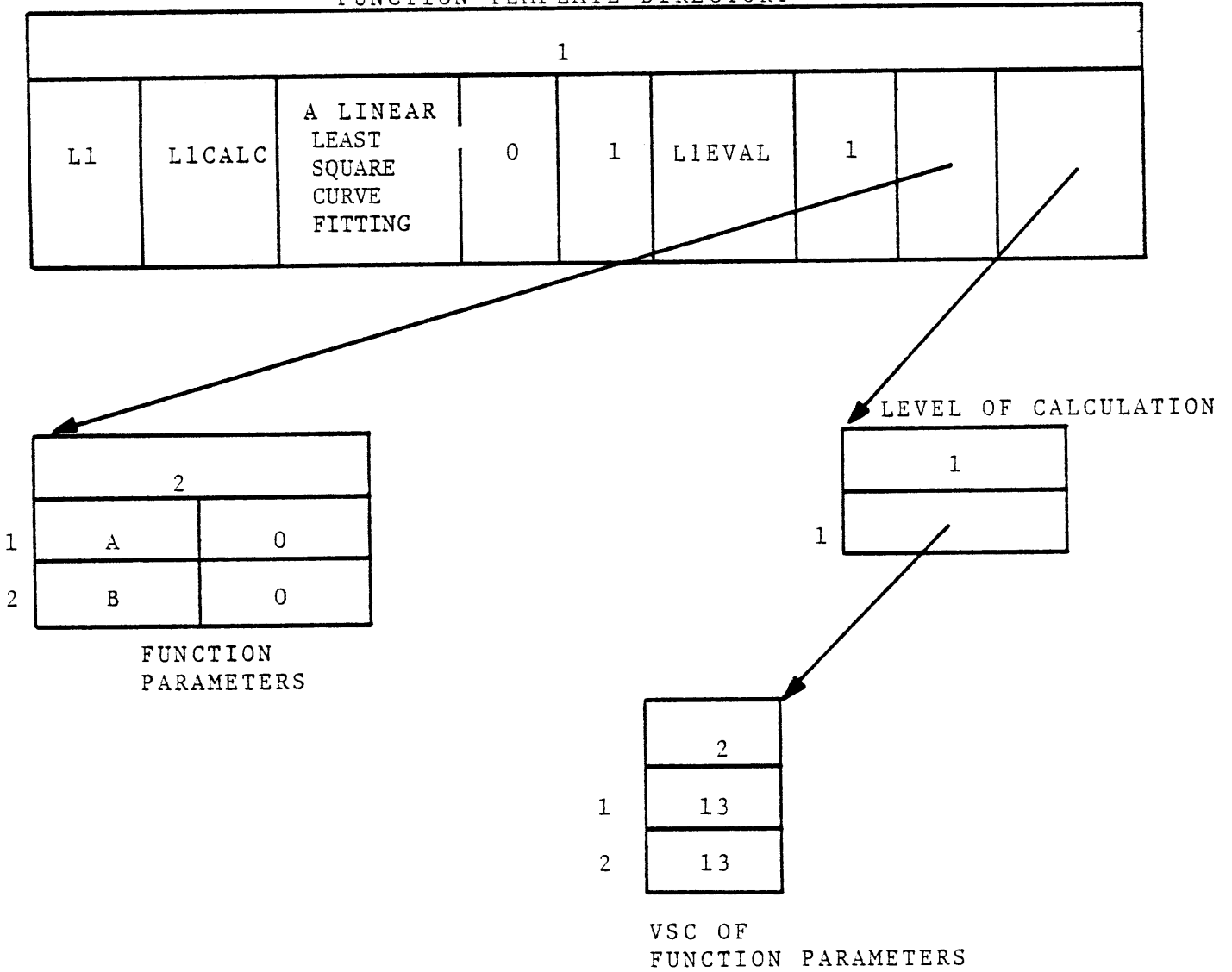


FIGURE 4.12 AN EXAMPLE OF A FUNCTION TEMPLATE

PROPERTY ESTIMATION METHODS TABLE

NUMBER OF PROPERTIES = N

	PROPERTY NAME	NUMBER OF OPTIONS	DEFAULT OPTION
1			
2			
N			

FIGURE 4.13 THE PROPERTY ESTIMATION METHODS TABLE STRUCTURE

PROPERTY ESTIMATION
METHODS TABLE

	4		
1	PVAP	2	1
2	FVAP	2	1
3	FLIQ	5	4
4	ALIQ	7	1

FIGURE 4.14 AN EXAMPLE OF A PROPERTY ESTIMATION
METHODS TABLE

CONTROL INFORMATION

SYSTEM (TBS)			
NAME		SERIAL NUMBER	COMPATIBILITY LEVEL
CONSISTENCY FLAG			
PROCEDURE TO CALCULATE ALL UNITS	PROCEDURE TO CALCULATE ALL COMPONENTS	PROCEDURE TO CALCULATE ALL FUNCTIONS	PROCEDURE TO CALCULATE ALL STREAMS
NUMBER OF LEVELS FOR ABOVE ROUTINE	NUMBER OF LEVELS FOR ABOVE ROUTINE	NUMBER OF LEVELS FOR ABOVE ROUTINE	NUMBER OF LEVELS FOR ABOVE ROUTINE
REFERENCE FOR ABOVE ROUTINE	REFERENCE FOR ABOVE ROUTINE	REFERENCE FOR ABOVE ROUTINE	REFERENCE FOR ABOVE ROUTINE
DEFAULT NUMBER OF SIGNIFICANT DIGITS			
DEFAULT NUMBER OF DECIMAL DIGITS			
DEFAULT DEBUGGING FLAG			

FIGURE 4.15 THE CONTROL INFORMATION STRUCTURE

different generations of the TBS. This number should be increased whenever the TBS is updated or expanded.

- c) the compatibility level of the TBS. There is such a number associated with each generation of the TBS. Those generations of a TBS having the same number are said to be compatible. The incompatibility problem may arise for a user dealing with two generations of a TBS. An example will illustrate the problem. Suppose a user of a TBS has created and saved a process model having among other elements streams of type "X". Then suppose that the TBS administrator has updated the TBS and has deleted the template of stream type "X". Now the user attempts to retrieve his saved process under the control of the updated TBS. Clearly his process model is not compatible with the current generation of the TBS. If this condition is not detected by the system, serious problems may arise. Although the above condition will be detected by the system, there are other similar conditions which the system either cannot detect or would require excessive computer time for detection. The compatibility level is a way for solving this problem. The TBS administrator should update the compatibility level whenever the updated template data base is not compatible with the previous generation. Extending

the data base (adding new templates) would not cause incompatibility. Deleting an already defined template as was the case in the above example will result in an incompatible TBS. Updating a previously defined template may or may not cause an incompatibility problem. For example, changing the number of phases of a stream type will result in an incompatible TBS. Other examples are given in Appendix B which describes the Template Definition Language.

- d) consistency flag indicating whether the template data base is internally consistent or not. For example, if the template of a stream type mentioned in a unit template is not defined, the data base is said to be inconsistent. Only consistent data can be said to define a TBS. The program "update_tdb" will automatically check the data base and set the flag, so that the inconsistent data base is not used by the GPES executive, thus avoiding unpredictable results.
- e) procedure to calculate all units or a ";". If such a routine has been implemented it would be called in response to the following user command: calculate unit all (optional argument list);
- The routine should be able to determine the order in which to calculate individual units. Such routines are discussed in more detail in Chapter 6.

- f) procedure to calculate all components or a ";".
- g) procedure to calculate all functions or a ";".
- h) procedure to calculate all streams or a ";".
- i) number of levels of calculation for each of the above routines.
- j) reference information for each of the above routines.
- k) default number of significant digits (between 1 and 14).
- l) default number of decimal digits (between 0 and 14).
- m) default debugging flag (0 or 1).

The last three items are initial (default) values of three of the five profile parameters. Profile parameters are those parameters that control the operation of PEL input and output commands. The profile parameters are:

sdigit: Significant number of digits to be used for printing numerical values (1 to 14).

ddigit: Decimal number of digits to be used for printing numerical values (0 to 14).

dflag: Debugging flag (between 0 and 3) which is used for debugging purposes. Deflag of zero which is for normal operation indicates that:

- a) the GPES executive should write a message on the user's terminal whenever it calls a calculating routine, and
- b) if a fatal error occurs in the calculating routine, the GPES executive should take over control and prevent the termination of the PEL session.

Dflag of one is a request for the latter only, and dflag of two which may be used by the TBS administrator is a request for the former only. The latter enables the TBS administrator or TBS programmers using the Multics debugging facilities to debug the TBS programs. Dflag of three which may be used by the GPES administrator for debugging the system is a request for additional information regarding the internal operation of the system. Default value of dflag can be set to either zero or one by the TBS administrator.

output: It specifies the user's output file which could be either his terminal or another file.

input: It specifies the user's input file which could be either his terminal or another file.

The PEL "profile" command enables the user to specify the profile parameters. The default values of the first three parameters (sdigit,ddigit,dflag) are those given in the control information. The default value of the input or output profile parameter is always the user's terminal. An example of control information is shown in Figure 4.16.

4.1.8. Text File

The text file is created and updated by any available editor. It contains the following information:

- a) The TBS administrator's name.
- b) Any reference about the TBS.
- c) Any messages to the TBS users.

The above information (a,b,c) will be printed when the user requests use of the TBS.

CONTROL INFORMATION			
MHD	1	1	
1			
;	;	;	;
-	-	-	-
-	-	-	-
6			
5			
0			

FIGURE 4.16 AN EXAMPLE OF A CONTROL INFORMATION

- d) The TBS lock, which indicates whether the TBS can or cannot be accessed by the users.
- e) Any news about the TBS. This information, along with similar information about the GPES will be printed in response to the user's "news" command.
- f) Any errors found or reported about the TBS. This information, along with similar information about the GPES, will be printed in response to the user's "bugs" command.

The format of the file is shown in Figure 4.17. Each of the above items should start and terminate by a "%". "tbs-name" indicates the TBS name. The format of each heading should be exactly as shown in the figure. The notation ----text--- indicates the text provided by the TBS system administrator. It can be any number of characters or lines, and may contain any character except "%", which terminates the text. The sequence of providing the above items is immaterial. The "lock" item should have one of the following formats:

```
%lock open%
```

```
%lock close%
```

If this item is not provided or it is as the first format, then the system can be accessed by the users. The second form indicates that the system cannot be accessed by the users. An example of a Text File is shown in Figure 4.18.

4.2 The Template Data Base Segments

The Template Data Base consists of a set of segments (files). These segments accommodate various data structures that form the template data base as listed in Table 4.2. Unit templates are stored in two segments: one contains the unit template directory, and the other contains all the

Figure 4.17. Format of the Text File

%*tbs-name system administrator:---text---%

%*tbs-name reference:---text---%

%*tbs-name message:---text---%

%lock open% or %lock close%

%tbs-name news:---text---%

%tbs-name bugs:---text---%

FIGURE 4.18 AN EXAMPLE OF A TEXT FILE

```
text      05/12/78  1322.2  edt  Fri

%*heat_exchangers system administrator:Sharif Arab-Ismaili
                                room 66-064, MIT, Cambridge, Ma,
                                telephone: (617) 253-6531 %
%*heat_exchangers reference:A simple heat exchangers network analyzer,
                                developed for testing the GPES %
%*heat_exchangers message:be good to yourself.%
%lock open%
%*heat_exchangers news: once upon a time there was.....%
%*heat_exchangers buss:you should be kidding....%
```

TABLE 4.2THE TEMPLATE DATABASE SEGMENTS

SEGMENT NAME	SEGMENT CONTENTS
1. utemp_dr	unit template directory
2. stemp_dr	stream template directory
3. ctemp_dr	component template directory
4. fntemp_dr	function template directory
5. dime_dr	dimension directory
6. estmeth_dr	property estimation methods table
7. unit_area	template substructures of unit types
8. strm_area	template substructures of stream types
9. comp_area	template substructures of component types
10. func_area	template substructures of function types
11. dime_area	dimension substructures
12. ctl_info	system control information
13. text	system administrator, reference, message, lock, bugs, news.

unit template substructures. Similarly, stream templates, component templates and function templates are each stored in two segments. Dimension table is also stored in two segments: one contains the dimension directory and the other dimension substructures. Property estimation methods table, control information, and text file each are stored in a separate segment.

The program "update_tdb" creates and manipulates all these segments, except the text segment which is created and manipulated by any available editor.

4.3 The Template Definition Language

The "update_tdb" is an interactive program for storing and updating the template data base. It can be used through a simple command language called Template Definition Language (TDL). The TDL command syntax is as follows:

```
command [object] [identifier]
```

The "object" and "identifier" are not always required. "command" is a language keyword indicating the function of the command, and "object" is another language keyword indicating the object upon which the action should take place. "identifier" is either another language keyword, an integer number, or a user supplied identifier representing the type of the object. For instance, if the system administrator wants to insert the template for a unit type pump the command for it would be:

```
insert unit pump
```

Certain abbreviated forms are also allowed, such as

```
i u pump
```

For the insert command, the program prompts the user for all the required information. The complete list of commands is given in Table 4.3.

TABLE 4.3

THE TEMPLATE DEFINITION LANGUAGE COMMANDS

COMMAND	OBJECT	IDENTIFIER
1. insert	unit	"type"
	stream	"type"
	component	"type"
	function	"type"
	dimension	"number"
	property	"number"
2. delete	unit	"type"
	stream	"type"
	component	"type"
	function	"type"
	dimention	"number"
	property	"number"
3. replace	unitlevel	"type"
	streamlevel	"type"
	complevel	"type"
	funclevel	"type"
4. print	unit	"type"
	stream	"type"
	component	"type"
	function	"type"
	dimension	"number"
	property	"number"
	dimtable	----
	proptable	----
	ctlinfo	----
	vsctable	----
	all	----
5. list	units	----
	streams	----
	components	----
	functions	----
6. revise	ctlinfo	sysname
		serialno
		compatlevel
		defsdigit
		defddigit
		defdflag
		unitall
		streamall
		compall
	funcall	
7. end	----	----

The function of each command is as follows:

insert	To insert the templates. The program prompts the user for all the required information. Once all the information has been provided, the program asks the user whether the template is to be inserted into the template data base, or to be ignored.
delete	To delete the templates.
print	To print the templates.
list	To list the existing types of units, streams, components or pre-defined functions.
replace	To replace a part of a template, associated with any one level of calculation. The program prompts the user for all the required information.
revise	To change an item of the control information. These items can never be deleted, only changed. The program prompts the user for all the required information.
end	Triggers the termination of the session.

Before the update_tdb program accepts any command, it asks the user whether the template data base is new or old. If the user's response is "old" the program prompts the user for the path name (path name is a Multics terminology indicating the address of a segment or a directory) of the directory containing the template data base segments. If the user's response is "new", the program prompts the user for the path name of the directory under which the template data base segments are to be created. Then the program creates and initializes the template data base segments. In particular, the program initializes the control information as follows:

```
sysname = ";"
```

```
serialno = 1
compatlevel = 1
defsdigit = 14
defddigit = 14
defdflag = 0
Procedure to calculate all units = ";"
Procedure to calculate all streams = ";"
Procedure to calculate all components = ";"
Procedure to calculate all functions = ";"
```

Note that ";" means that the routine is not implemented. The user (TBS Administrator) should use the revise command to specify the TBS name (sysname) or to change the above default values.

During the session, the system administrator can exit from the program by pushing the "quit" button on the terminal. Reentry after such an abnormal exit can be accomplished by using the "pi" (for Program Interrupt) command of Multics; this causes the current command activities to be ignored and the program becomes ready to accept a new command. The use of this mechanism is only recommended while providing information for an insert or replace command when some errors has been discovered in previously supplied information or to abort the printout of a print command. Use of this mechanism under other conditions may damage the data base.

Upon issuance of the "end" command, the program proceeds to check the consistency of the data base. The consistency aspects are discussed in section 4.4.1. If there are any inconsistencies, the system administrator may want to return to command level to enter more commands. He or she is therefore asked whether he or she would like to exit. If the answer is

"yes", the program asks for the default types of units, streams, components and functions and these are made the first members of their respective directories. When the GPES Executive wants a default type, it looks for the first member of the corresponding directory.

The TDL language has been designed to be very simple. This characteristic often makes it very inflexible. For example, to change an item of information about a template one may have to delete that template and insert a new one with the updated information. Only items of control information can be changed directly. Information regarding any level of calculation for a unit, stream, component, or function template can also be replaced without deleting the entire template. A detailed description of each TDL command is given in Appendix B. A sample computer session with TDL is shown in Figure 10.12. The template printouts presented in Chapter 10 are also the result of TDL print commands.

4.4 "update tdb" Program

The sole purpose of the program is the management of data stored in the template data base. The data stored in the data base must of course, be internally consistent. The program "update_tdb" therefore, must check for the consistency of the data, as will be described in the next section.

A schematic flowchart of the structure of the program is shown in Figure 4.19. For a new data base the program creates and initializes the data base segments. The program accepts one command at a time. The command is processed either by an internal procedure, or by the external procedure, print-temp, which is also used by the GPES executive for printing information in the Template Data Base. The structure of the program, in this sense, is modular, this is expected to facilitate the maintenance of the program, which otherwise would have been difficult

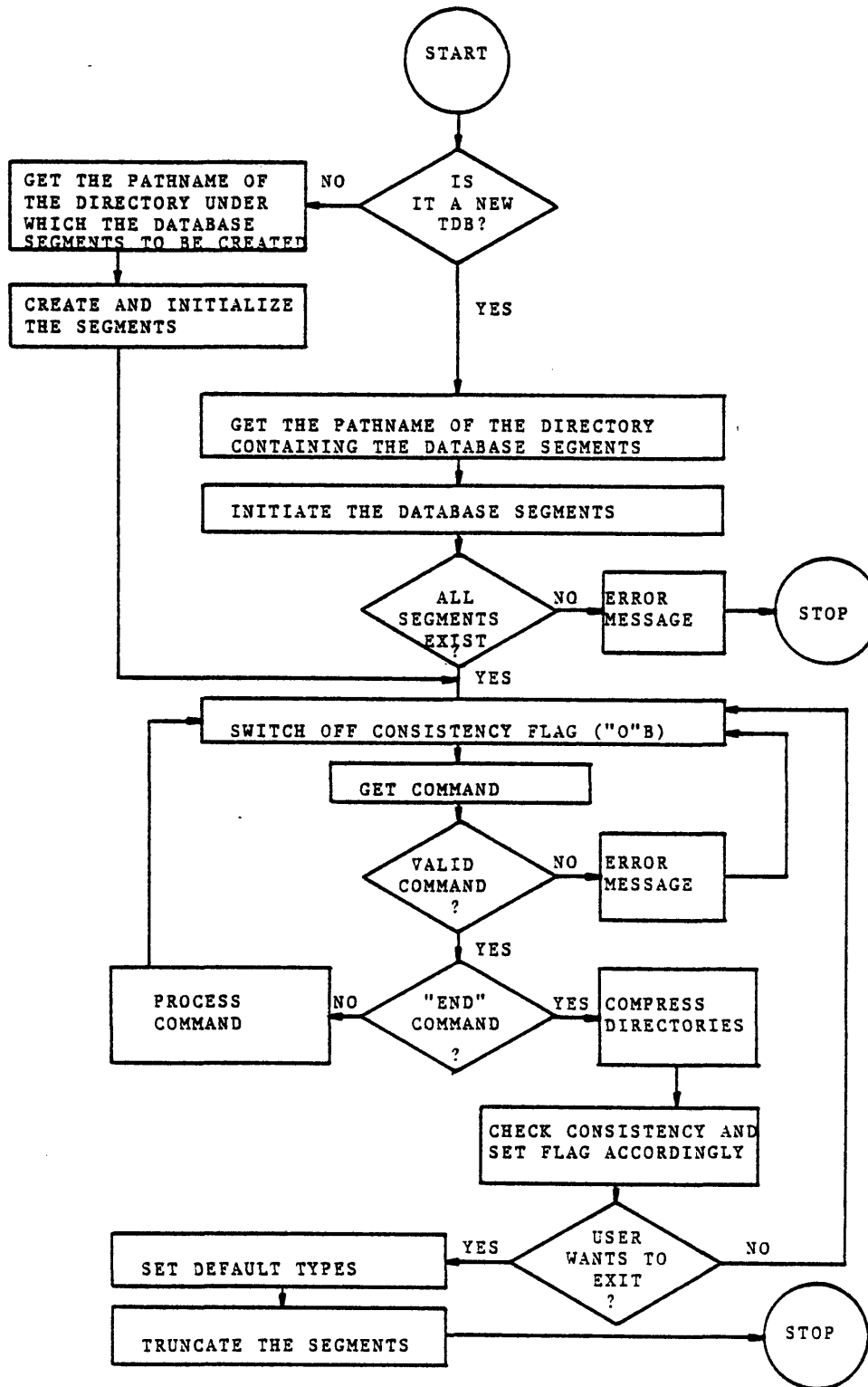


FIGURE 4.19 THE FLOWCHART FOR UPDATE_TDB PROGRAM

considering its size. The program creates temporary structures to contain information provided by the user for insert and replace commands. Once all the information is provided and the user approves its correctness, permanent structures will be created and filled by the supplied information. The temporary structures are then deleted. The purpose of the temporary structures is to improve the consistency aspect of the data base, and to allow the implementation of the "quit" mechanism as described earlier.

4.4.1 Consistency of the Data in the Template Data Base

The data stored in the data base must, of course, be internally consistent. Only consistent data can be said to define a template-based system. The program `update_tdb`, therefore, checks to see if the following conditions are true:

1. All stream types mentioned in the unit templates must exist in the stream template directory.
2. All component types mentioned in the stream templates must exist in the component template directory.
3. All dimension types mentioned in all parameter templates must exist in the dimension table.
4. There should be no undefined dimension number less than the maximum defined number. For instance, if dimension number 9 is defined, so should the numbers 1 to 8. This is to ensure that no entry of the dimension table is empty.
5. The above also holds true for the property estimation methods table.
6. The TBS should have a name.
7. Default number of decimal digits should not be greater than the default number of significant digits.

These checks are made before the termination of the session. During the session, while the data base is being updated, the program does not allow clearly invalid data to be inserted. For instance, it will not allow a character string where a number is required. It will not allow a value status code to be less than 1 or greater than 15, and so on.

If there is any inconsistency, the consistency flag is turned off, so that the data base is not used by the GPES Executive, thus avoiding unpredictable results.

During the session, the system administrator can exit from the program by pushing the "quit" button on the terminal. This abnormal exit can leave the data base in an inconsistent state. To protect against this, the consistency flag is turned off at the beginning of the command processing loop. Reentry after an abnormal exit can be accomplished by using the "pi" (for program interrupt) command of Multics, that returns control to the top of the command processing loop.

4.5 Protection of the Template Data Base

Two copies of every Template Data Base are required and a third one is recommended. The two required copies are:

1. The original, or primary copy accessible by the TBS Administrator. This copy is created and manipulated by the update_tdb program.
2. The system copy, accessible to the GPES Executive. The segment names for this copy are the same as in Table 4.2 except that they are suffixed by ".syscopy". The system copy can be used while the primary copy is being updated. The users have access to this copy only.

One or more backup copies are recommended. They may be used to retain the previous generations of the Template Data Base and/or to contain the current generation to enable recovery from unintentional erasures and other accidents. The segment names for each of these copies are the same as in Table 4.2, except that they are suffixed by a symbol chosen by the TBS administrator. Suffixes such as ".BACKUP", ".DATE", or ".SERIAL_NO", are recommended. The flow of information between the different copies of the Template Data Base, update_tdb, and GPES Executive is shown in Figure 4.20. The copy_tdb is a utility program which copies one set of the data base into another. Utility programs are described next.

4.5.1 Utility Programs

As has been discussed, the update_tdb program creates and updates the original copy of the Template Data Base, except the text file. The original copy of the text file is created and updated by any available editor. Other utility programs have been developed to facilitate the TBS Administrators' tasks, as listed in Table 8.4. These utility programs are command procedures (a program composed of a set of Multics commands) and are called by Multics command "exec_com". A description of each program will follow.

4.5.1.1 gaccess tdb

Although the TBS administrator should have read and write access to all copies of the Template Data Base, the users should have only read access to the system copy of the data base (this is for protection of the data base). The command procedure gaccess_tdb gives a user read access to the system copy of the Template Data Base. It requires three arguments and is called as follows:

```
ec gaccess_tdb Directory Person_ID Project_ID
```

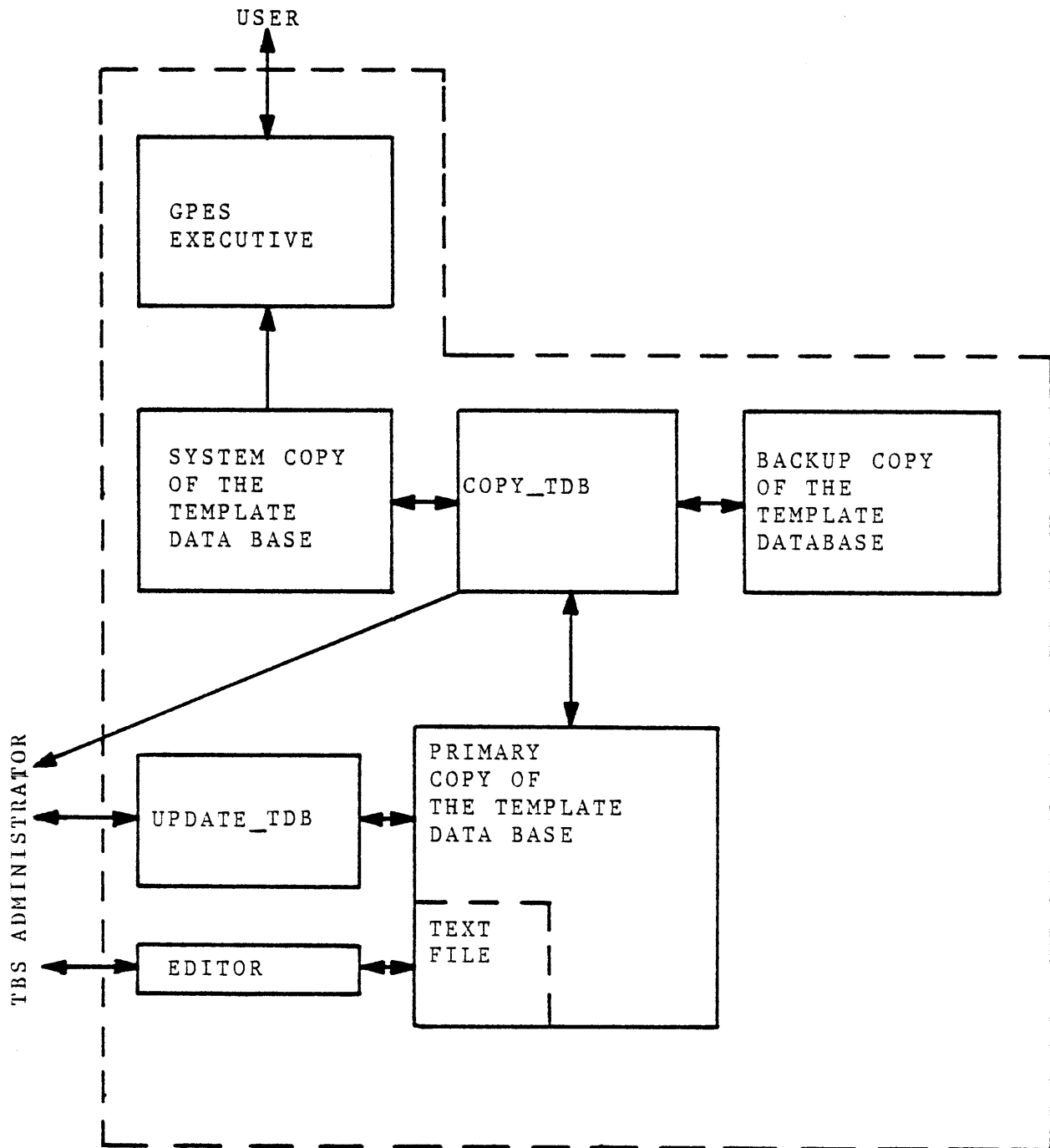


FIGURE 4.20 INFORMATION FLOW REGARDING THE TEMPLATE DATA BASE.

Where:

Directory Is the relative or absolute path name of the directory containing the Template Data Base segments.

Person_ID Is the user's Multics Person ID. Using a "*" in this field indicates all users of the given project.

Project_ID Is the user's Multics Project ID. Using a "*" in this field indicates all projects for the given user.

Example

```
ec gaccess_tdb >udd>ICPES>Arab-Ismaili>heat_exchangers * MHD
```

will give read access to a system copy of the Template Data Base (located under the specified directory) to all users of project MHD.

The TBS Administrator should write a similar command procedure to give execute access to all TBS programs (calculating routines, etc.) to a user. The TBS administrator should also register each user with the GPES administrator by filling out the form shown in Figure 8.5, and sending it to the GPES administrator.

4.5.1.2 taccess_tdb

This command procedure is used to remove user access to a Template Data Base. It is called as follows:

```
ec taccess_tdb Directory Person_ID Project_ID
```

Where the three arguments are the same as described for gaccess_tdb. The TBS administrator should write a similar command procedure to remove user access to TBS programs (calculating routines, etc.). To remove a user from the list of authorized users of the TBS, the TBS administrator should also notify the GPES administrator by filling out and sending the form shown in Figure 8.5.

4.5.1.3 copy_tdb

This command procedure is used to copy a Template Data Base into another Template Data Base, while preserving the access rights of the target data base. It is called as follows:

```
ec copy_tdb Directory Suffix1 Suffix2
```

Where Directory is the relative or absolute path name of the directory containing the Template Data Bases. Suffix1 is the Suffix of the Data Base to be copied from, and Suffix2 is the Suffix of the Data Base to be copied into. Examples of the command are:

```
a) ec copy_tdb heat_exchangers "" .syscopy
```

will copy the primary Template Data Base (TDB) into the system copy. If the system copy does not exist a new one will be created and the TBS administrator is the only one who will have access to it. If the system copy does exist the copying operation will preserve its access list.

```
b) ec copy_tdb heat_exchangers .backup .syscopy
```

will copy the backup copy of TDB into the system copy.

```
c) ec copy_tdb heat_exchangers .backup ""
```

will copy the backup copy of TDB into the original copy.

4.5.1.4 copy_seg

This command procedure is used to copy one segment into another, while preserving the access rights of the target segment. It is called as follows:

```
ec copy_seg Directory segment1 segment2
```

where Directory is as defined earlier, segment1 is the name of segment to be copied from, and segment2 is the name of segment to be copied into.

Although the TBS Administrator may use this command procedure for copying any segment, it is recommended only for copying the text segment,

if that is the only segment in which some changes has been made. For example, the following commmand will copy the primary text segment into the system copy:

```
ec copy_seg heat_exchangers text text.syscopy
```

4.5.1.5 delete_tdb

It is used to delete a TDB. It is called as follows:

```
ec delete_tdb Directory Suffix
```

where directory is as defined earlier, suffix is the suffix of the TDB to be deleted. For example, to delete the backup copy of a Template Data Base:

```
ec delete_tdb heat_exchangers .backup
```

CHAPTER 5

DATA STRUCTURES REPRESENTING A PROCESS FLOWSHEET

GPES differs from most computer simulation systems in its representation of the Process Flowsheet. Process elements such as units, streams, components, etc. are represented by data structures connected to form a network. This network represents the process flowsheet and is created and manipulated by the GPES Executive in response to the user's commands. As the user concept of the process flowsheet changes, the network of data structures is manipulated to reflect those changes. This will permit the user to retain an active model of the problem being solved. The resulting model may be saved indefinitely for later modification and analysis.

GPES employs generalized data structures to represent process elements. However, before the system can utilize them to model a process configuration, it requires certain information about the types of units, streams, components, etc. As discussed in the previous chapter, this information is stored in the form of templates in the Template Data Base.

5.1 Memory Management

The data structures representing a process flowsheet are dynamically allocated when they are needed and will be freed when they are not required. Data structures of this kind are called "based data structures" in PL/1 terminology. The process network may be saved by the user for future retrieval and modifications. Hence, these data structures would be allocated in "areas" (a PL/1 terminology) for simplicity of saving and retrieving the process. Since the storage requirements for these structures vary for each process flowsheet, storage allocation strategy may

be influenced by the desire to increase user flexibility or system efficiency. In this section several methods are presented.

A) Single partition allocation with fixed size: In this approach a single fixed size storage area will be allocated for a process. Process data structures will be allocated in this area and they are accessed by offset locators (an offset locator designates the location of a data structure within the storage area).

Advantages:

- 1 - Simple and efficient for the system.

Disadvantages:

- 1 - The size of the partition is fixed regardless of the space needed by the problem being solved.
- 2 - The process size cannot exceed the size of the partition.

B) Single partition allocation with variable size: It is the same approach as the first method with the exception that the size of the partition should be specified by the user.

Advantages:

- 1 - Relatively simple and efficient for the system.

Disadvantages:

- 1 - User must provide the size of the partition he needs for his model.
- 2 - The maximum partition size is limited (for IBM PL/1 it is 32,766 Bytes, for Multics it is about 1,048,000 Bytes).

C) Dynamic partition allocation: In this approach there may be more than one partition made available. As the process grows and the need for additional space is recognized, another partition will be allocated dynamically. AREA CONDITION signals the need for additional space. In

this approach the addressibility is achieved by the pair of partition-number and offset in that partition. The size of the partition should be selected based on the system efficiency and storage requirement for average jobs.

Advantages:

- 1 - Allocates partitions as needed.
- 2 - There is no limitation on the size of the problem being solved.

Disadvantages:

- 1 - System overhead associated with two level addressing.

The latter approach meets the requirements of GPES and has been implemented. To allocate data structures within partitions, the GPES Executive starts from the last allocated partition and if AREA CONDITION is raised it tries the previous partition. If the required storage is not found in the available partitions a new partition will be allocated to meet the requirements. The collection of these partitions is called the working area. It should be noted that, once a partition is allocated, it will not be freed before the end of a process analysis or the specification of the "clear" instruction by the user. In this implementation of the system (Multics Implementation) the size of the partition is varying. Initially, when it is allocated it occupies only one page of storage (about 4,000 Bytes in Multics). As the need for additional storage is recognized the partition size automatically grows until it reaches the maximum of 255 pages (about 1048,000 Bytes), then another partition is allocated. A partition table keeps track of the locations of the partitions. A partition table is a part of the process directory data structure which is described next.

5.1.1 The Process Directory Data Structure

The directory data structure which is shown in Figure 5.1 contains the following information:

- a) The name, serial number and compatibility level of the system (i.e., GPES) and TBS under which the process was first created.
- b) Purity Flag, indicating whether the process has or has not been accessed by either an incompatible version of the system or TBS, or by a different TBS.
- c) The date and time the process was first created.
- d) A pointer to the beginning of the unit structure's list.
- e) A pointer to the component directory.
- f) A pointer to the beginning of the pre-defined function structure's list. There is no similar information in the directory for the user-defined functions, because user-defined functions are not allocated in the working area, as will be described later in this chapter.
- g) Two pointers to the beginning of the stream structure's list. This redundancy is required due to the internal structure and organization of the GPES executive.
- h) A pointer to the beginning of the variable structure's list.
- i) A pointer to the data structure containing the "property estimation methods in use".
- j) The current number of allocated partitions.
- k) A pointer to each allocated partition.

P_DR

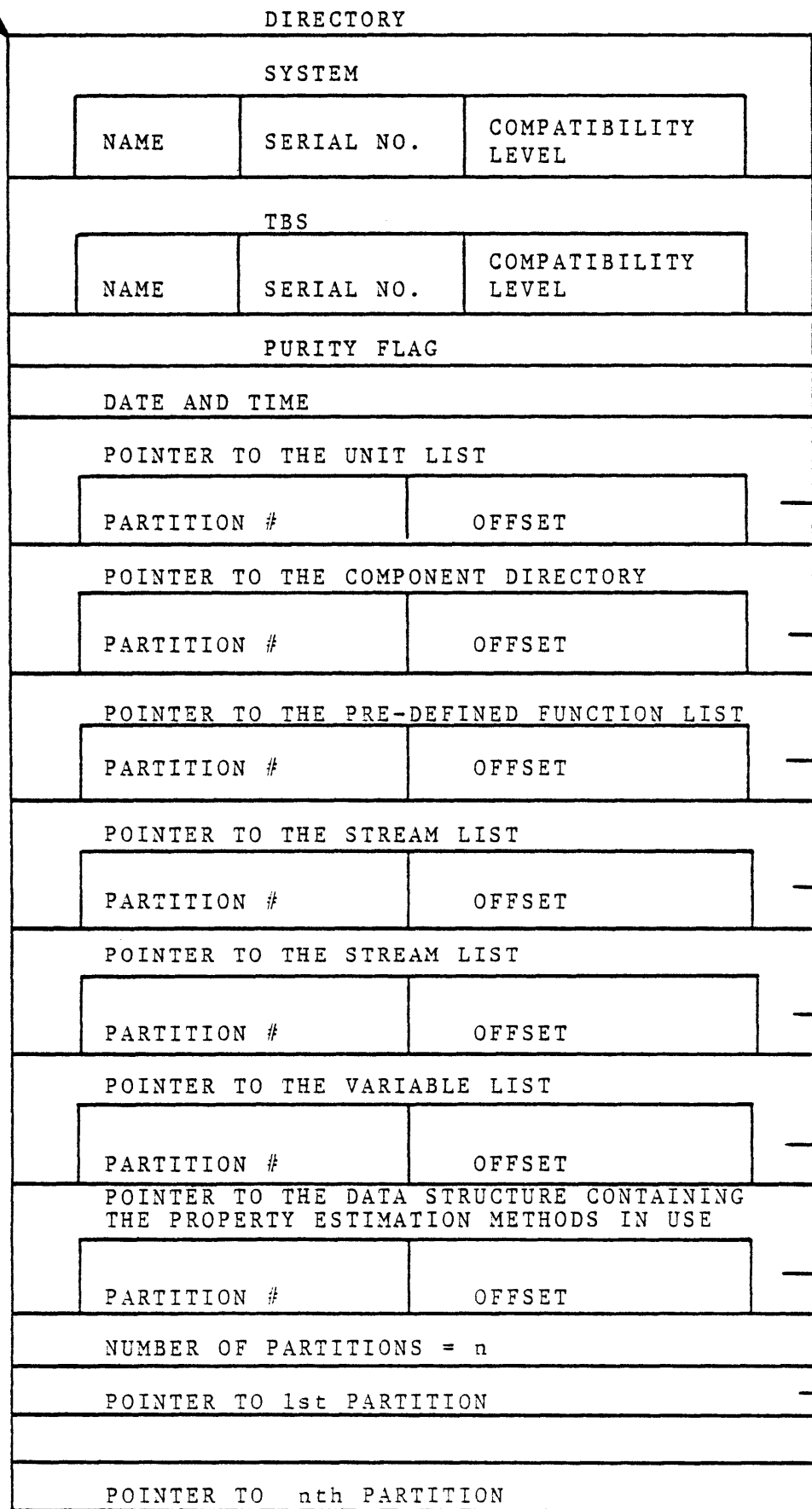
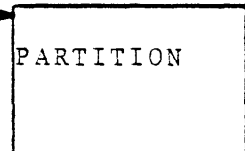


FIGURE 5.1
THE PROCESS
DIRECTORY
DATA STRUCTURE



Every piece of information about the process either is contained in the process directory or can be obtained by navigating the list of data structures which are chained to the directory. The external variable "p_dr" points to the directory, and therefore makes the directory accessible to the routines requiring such information. An example of a process directory having only one partition is shown in Figure 5.2.

5.2 Data Structures Representing Process Elements

A description of various data structures representing the process elements is given. The parameters structure which is common to most of these elements is described first.

5.2.1 Parameters Structure

The parameters structure as shown in Figure 5.3 is used to contain a set of parameters. A set of parameters may represent any of the following:

- 1) parameters of a unit.
- 2) parameters of a phase of a stream.
- 3) flow parameters of a component in a phase of a stream.
- 4) parameters of a component.
- 5) parameters of a pre-defined function.

Therefore, each unit has a parameter structure to contain its parameters. A stream may have many parameter structures to contain phase and flow parameters of each phase. A component or a pre-defined function each have a parameter structure to contain their parameters. Each parameter in the parameter structure is represented by a value and a value type. The value type indicates how the value has been assigned.

The value type of a parameter can be one of the following:

1. Unspecified (value type = 0). When no value has been assigned or user has explicitly unspecified the parameter by an unspecify command, the parameter is said to be unspecified.

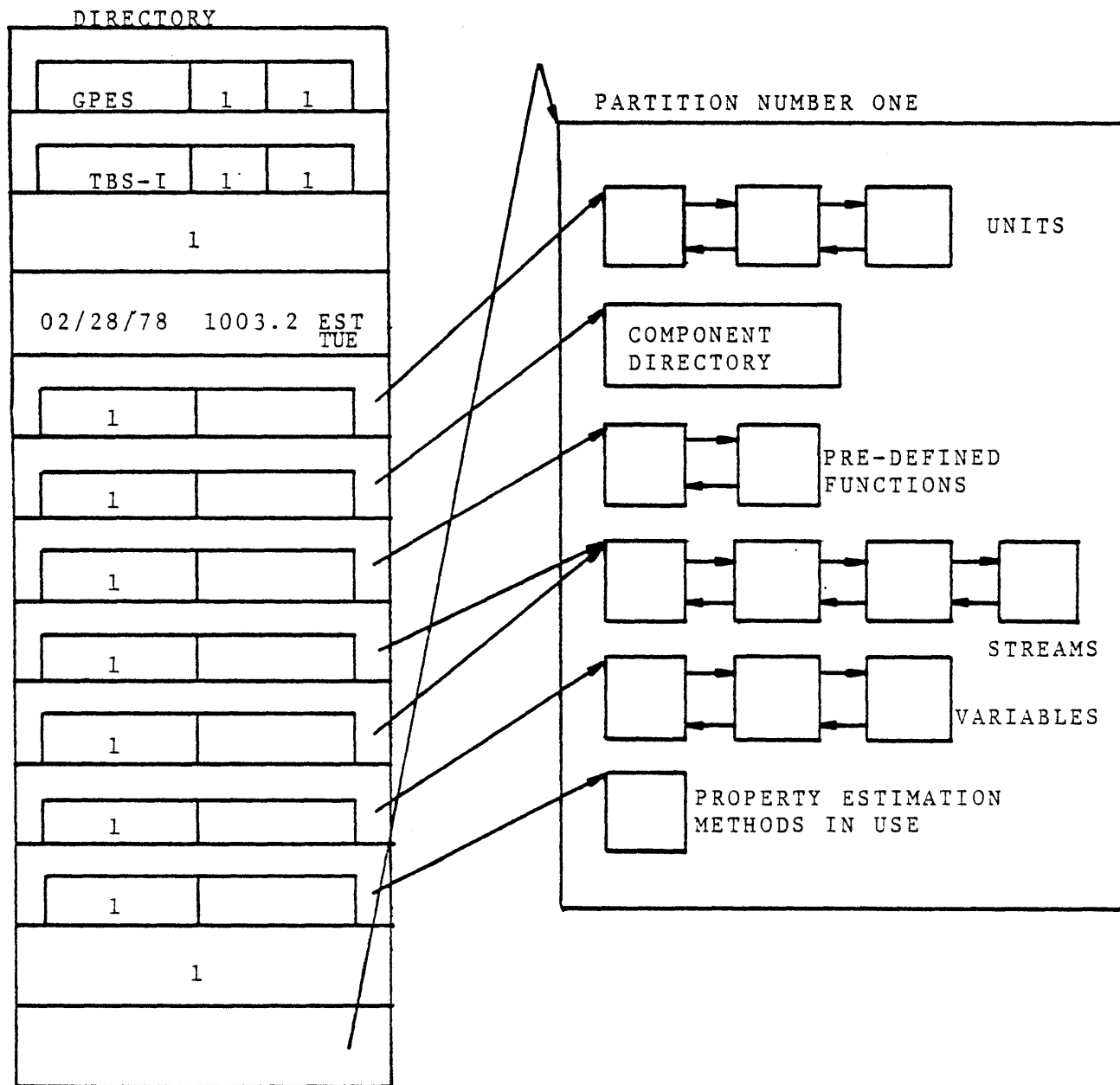


FIGURE 5.2 AN EXAMPLE OF A PROCESS DIRECTORY DATA STRUCTURE

PARAMETERS STRUCTURE

NUMBER OF PARAMETERS = N		
1	VALUE	VALUE TYPE
2		
N		

FIGURE 5.3 THE PARAMETERS STRUCTURE

2. Assumed (value type = 1). The value has been assigned by one of the following commands: assume, leta, or reada. The user assumes a value for a parameter if the exact value of the parameter is unknown, but a value is required to initiate calculation.
3. Specified (value type = 2). The value has been assigned by one of the following commands: specify, let, read, or repeat.
4. Calculated (value type = 3). The value has been assigned by a calculating routine.

The value type serves the following purposes:

1. To indicate a value has been assigned to a parameter.
2. To avoid calculating a parameter which has been fixed (specified) by the user.
3. To facilitate error checking for under- or over-specification of the parameters for a calculating routine.
4. To indicate which parameters are the output of a calculating routine, and hence facilitate the debugging effort.

5.2.2 The Unit Structure

The data structure representing a process unit is shown in Figure 5.4. Unit data structures are linked together by two sets of pointers. Each unit points to the succeeding unit and also points to the preceding unit. The pointer to the beginning of the chain of units is kept in the process directory data structure. The positions of the unit data structures in the chain are not the same as the position of corresponding units in the process flowsheet. When a new unit data structure is created, it would be added to the beginning of the chain, and it will remain as the first unit in the chain until a) it is deleted or b) another unit data structure is created.

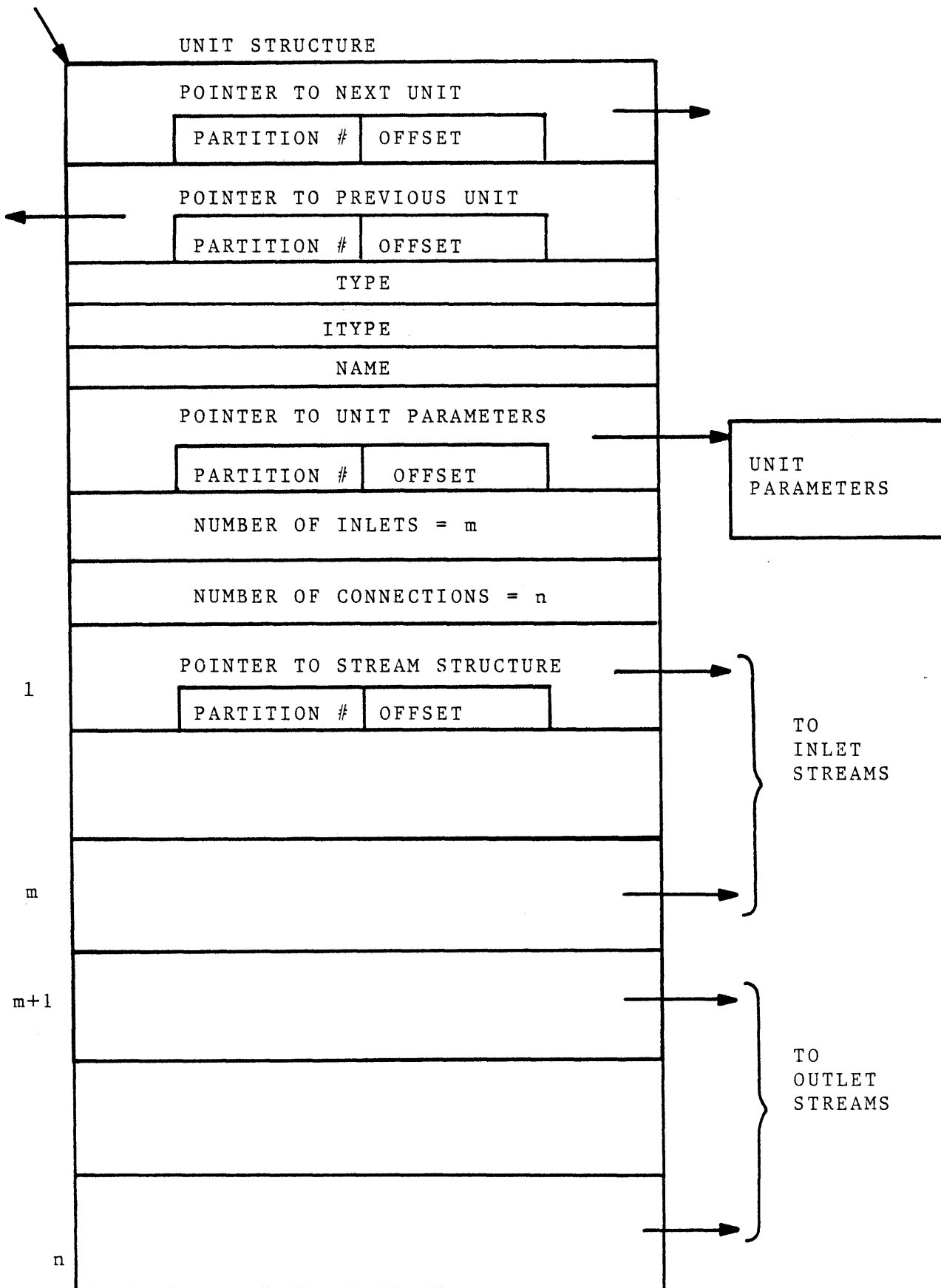


FIGURE 5.4 THE UNIT STRUCTURE

The unit data structure also contains:

- a) Unit type (TYPE) and the index of the unit type in the unit template directory(ITYPE). The ITYPE is provided to avoid searching the unit template directory each time it is required to access the unit's template. Whenever a process which is saved earlier is retrieved, the system will update the ITYPE. This is a necessary action because it is foreseeable that the TBS system administrator may have updated the template data base and consequently may have caused the reordering of the unit templates in the directory. It is for this reason that both TYPE and ITYPE have been provided.
- b) Unit's name.
- c) A pointer to a parameter structure which contains the unit parameters. The parameter structure is created at the same time the unit's structure is created. The value of each parameter is unspecified until a value has been assigned by the user or by a calculating routine.
- d) Number of inlets. This number is taken from the unit template.
- e) Number of connections. This number is also taken from the unit template. It is the total number of inlets and outlets.
- f) For each connection a pointer to the corresponding stream's structure. This pointer is null (partition_no =0) when no stream is connected to the unit. These entries are updated in response to the following user's requests:
 - i) Connect the unit to some streams,
 - ii) Disconnect the unit from some streams,

- iii) Delete the streams which already have been connected to the unit.

An example of a unit structure is shown in Figure 5.5. The unit template was shown in Figure 4.8.

5.2.3 The Component Structure

The information regarding all components is stored in a data structure called the component directory as shown in Figure 5.6. Each entry of the directory corresponds to a chemical component. The number of entries of the directory which is the maximum number of components that can be present in the process is set by the user. Before the system accepts any command, it prompts the user for this number and creates such a data structure. All entries, of course, are empty at this time. When the user creates a component or loads a component from a component file, the first non-empty entry of the directory will accommodate that component. When the user deletes a component, that entry again becomes empty and a possible location for a new component that may be created later. Each entry contains the following:

- a) Component name.
- b) Number of references. This is the number of streams that contain the component. It is used to protect users from deleting components which are present in some streams. This number is initially zero. The number will be incremented by one (1) whenever the component is said to be present in another stream. Similarly, it will be decremented by one (1) for each stream that no longer contains the component.
- c) Component type.
- d) Index of the component type in the component template directory.

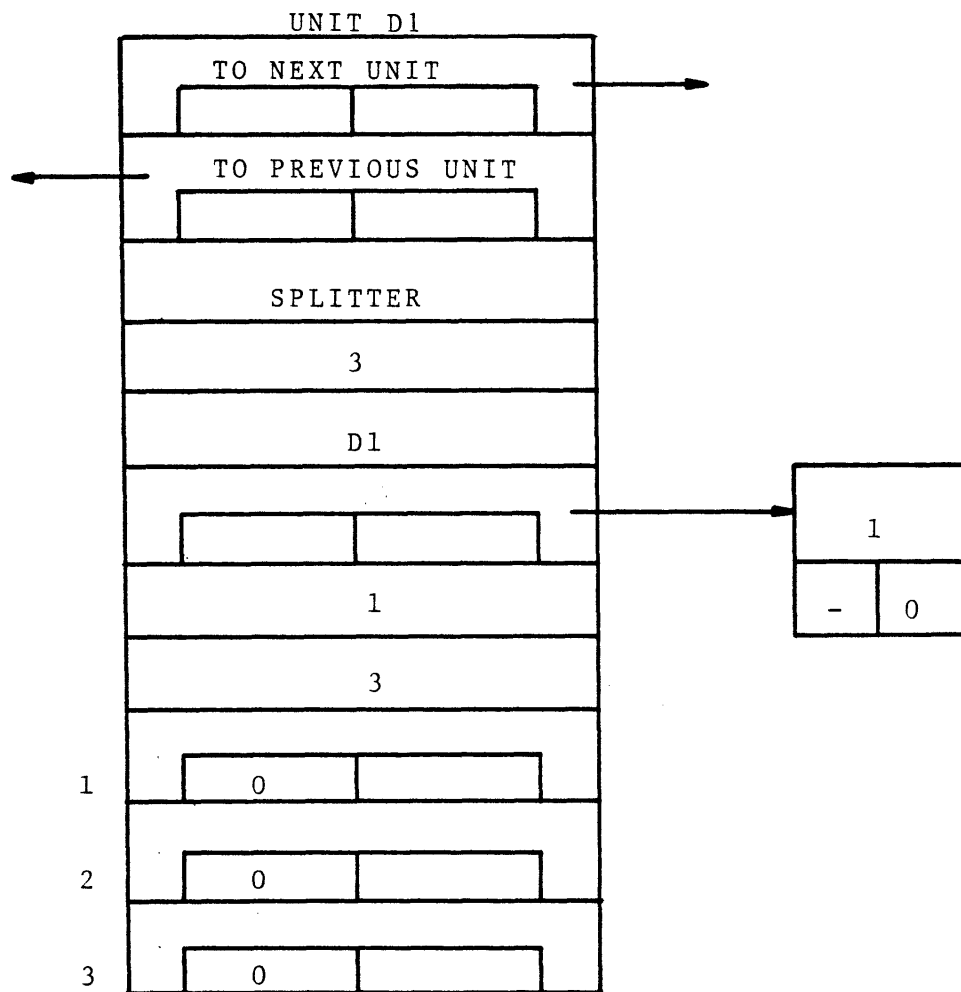


FIGURE 5.5 AN EXAMPLE OF A UNIT STRUCTURE

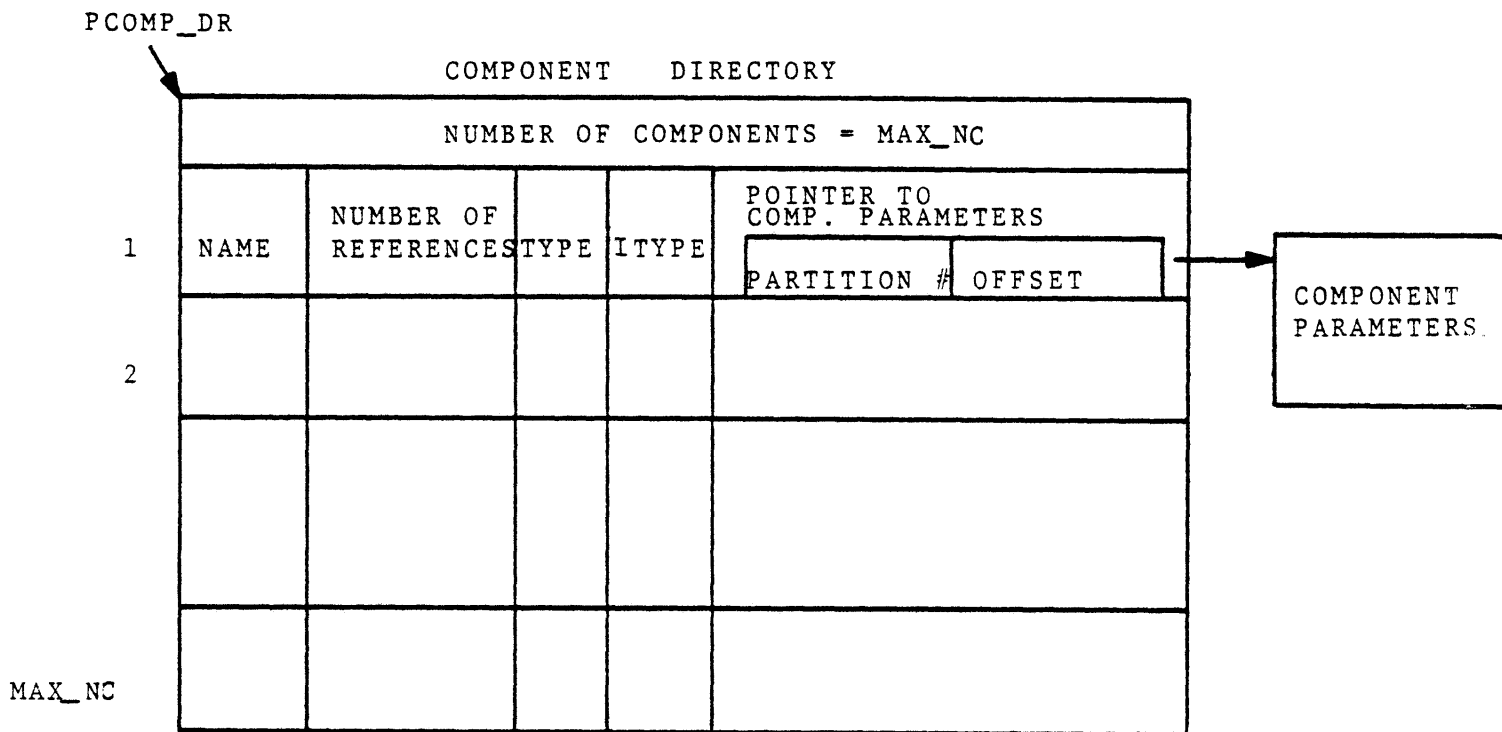


FIGURE 5.6 THE COMPONENT DIRECTORY STRUCTURE

- e) A pointer to a parameter structure which contains the component's parameters. For an empty entry it is a null pointer.

Storing the components in a component directory as described above imposes the following disadvantages:

- a) The user has to specify the maximum number of components in a process.
- b) The above number is fixed for the entire life of the process.

Although a mechanism could be provided to remedy the above disadvantages, it is a very time-consuming process. Once the number of components exceeds the pre-specified maximum number, it requires that a new directory with a larger number of entries be created and the old directory be copied into the new directory and then be deleted. This will force recreation of all streams which in turn forces the updating of all units. Therefore, this mechanism has not been implemented. Instead the user is recommended to use his best judgment in providing the maximum number of components.

Although the representation of components by a chain of data structures similar to the one described for units does not have the above disadvantages, it has the following more serious disadvantages:

- a) The representation of streams would become more sophisticated in order to refer to components which are flowing in them.
- b) TBS Programs (Calculating Routines) need to be sophisticated in dealing with components.

On the other hand, the component directory will not impose the above disadvantages. Hence, it has been used for representing the components. The entry number of a component in the directory is known as its component-index. Figure 5.7 shows an example of a component directory having two components.

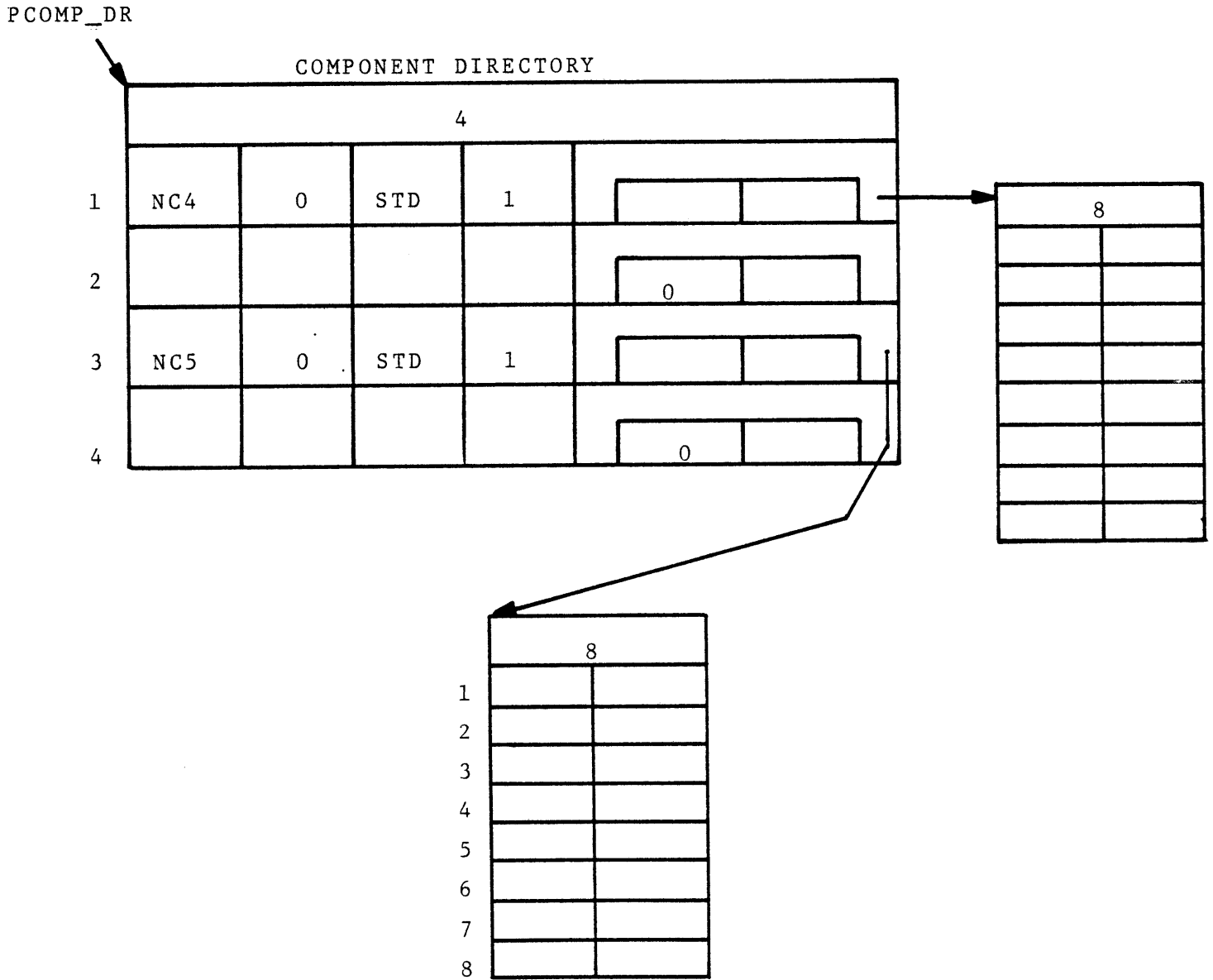


FIGURE 5.7 AN EXAMPLE OF A COMPONENT DIRECTORY

5.2.4 The Stream Data Structure

A stream data structure as shown in Figure 5.8 contains the following:

- a) A pointer to the next stream's data structure.
- b) A pointer to the previous stream's data structure.
- c) Stream type.
- d) Index of the stream type in the stream template directory.
- e) Stream name.
- f) A pointer to the source unit and the connection number of the unit where the stream originates.
- g) A pointer to destination unit and the connection number of the unit where the stream enters. These two pointers are updated in response to the following user commands:
Connect or disconnect commands,
Delete the source or destination unit.
- h) Number of phases. This is in addition to Phase 0 which represents the total stream.
- i) Maximum number of components (max_nc).
- j) For each phase the following:
 - 1) A pointer to a parameter structure which contains the phase parameters.
 - 2) "max_nc" entries, where each entry represents the flow of a component in the stream. The entry contains a pointer which points to a data structure which contains the flow parameters of that component in the phase. The pointer is null if the corresponding component is not present in the stream.

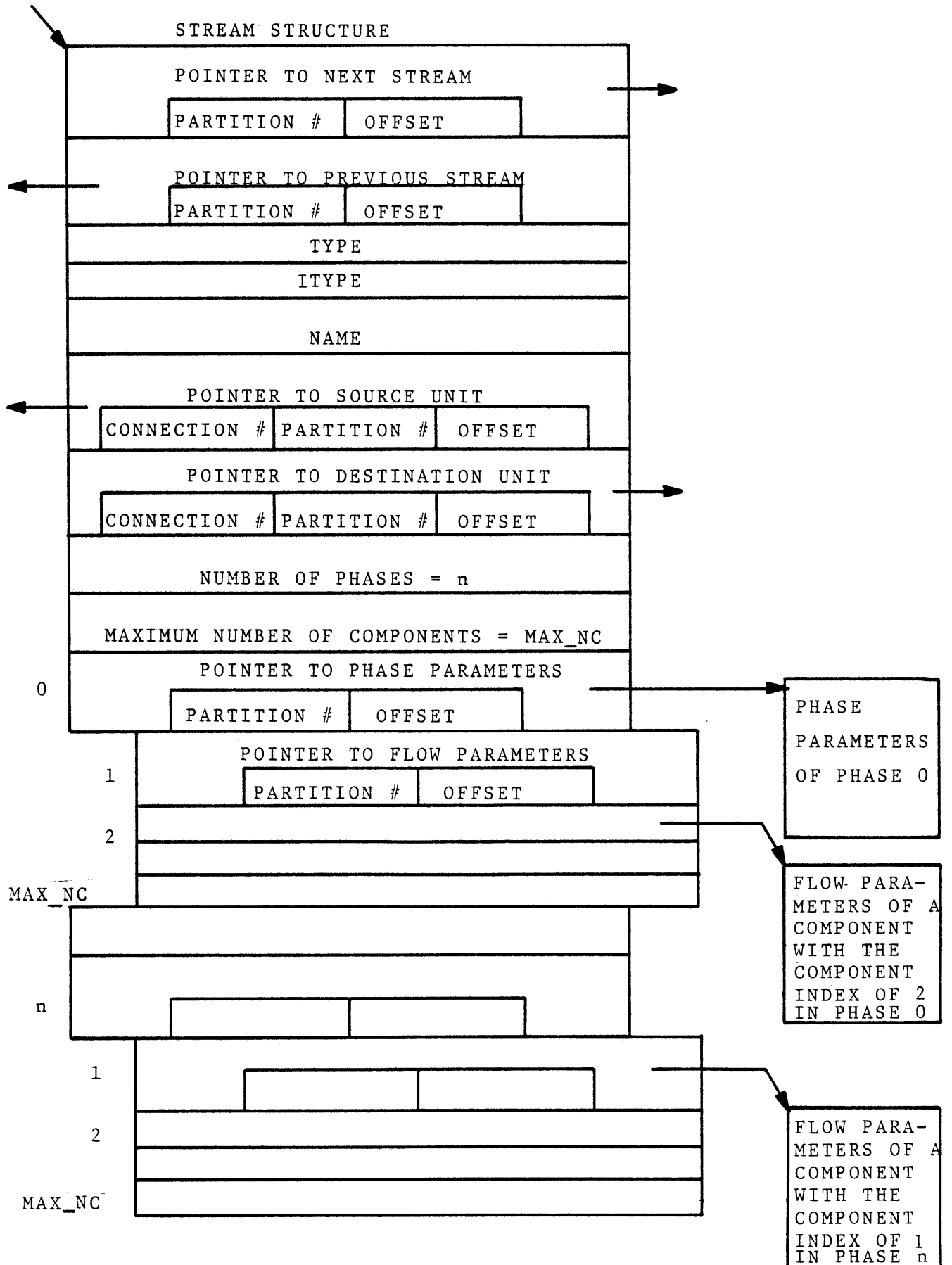


FIGURE 5.8 THE STREAM STRUCTURE

Figure 5.9a shows a stream data structure of type "STD" which was defined in Figure 4.6. Figure 5.9B shows the same stream containing components NC4 and NC5.

5.2.5 Pre-Defined Function Data Structure

The pre-defined function data structure is shown in Figure 5.10. It contains the following:

- a) A pointer to the next pre-defined function.
- b) A pointer to the previous pre-defined function.
- c) Function type.
- d) Index of the function type in the function template directory.
- e) Function name.
- f) Number of references. This is the number of times the existing user-defined functions have referred to this function. It is to protect users from deleting a pre-defined function which is already referred to by the existing user defined functions. In other words, a pre-defined function having a positive number of references cannot be deleted.
- g) A pointer to a parameter structure which contains the function parameters. These parameters are initially unspecified.

The data structure for a pre-defined function defined in Figure 4.12 is shown in Figure 5.11.

5.2.6 Data Structures for a User-Defined Function

The general form of the data structure representing a user-defined function is shown in Figure 5.12. It contains the following:

- a) A pointer to the next user-defined function.
- b) A pointer to the previous user-defined function.
- c) Function name.

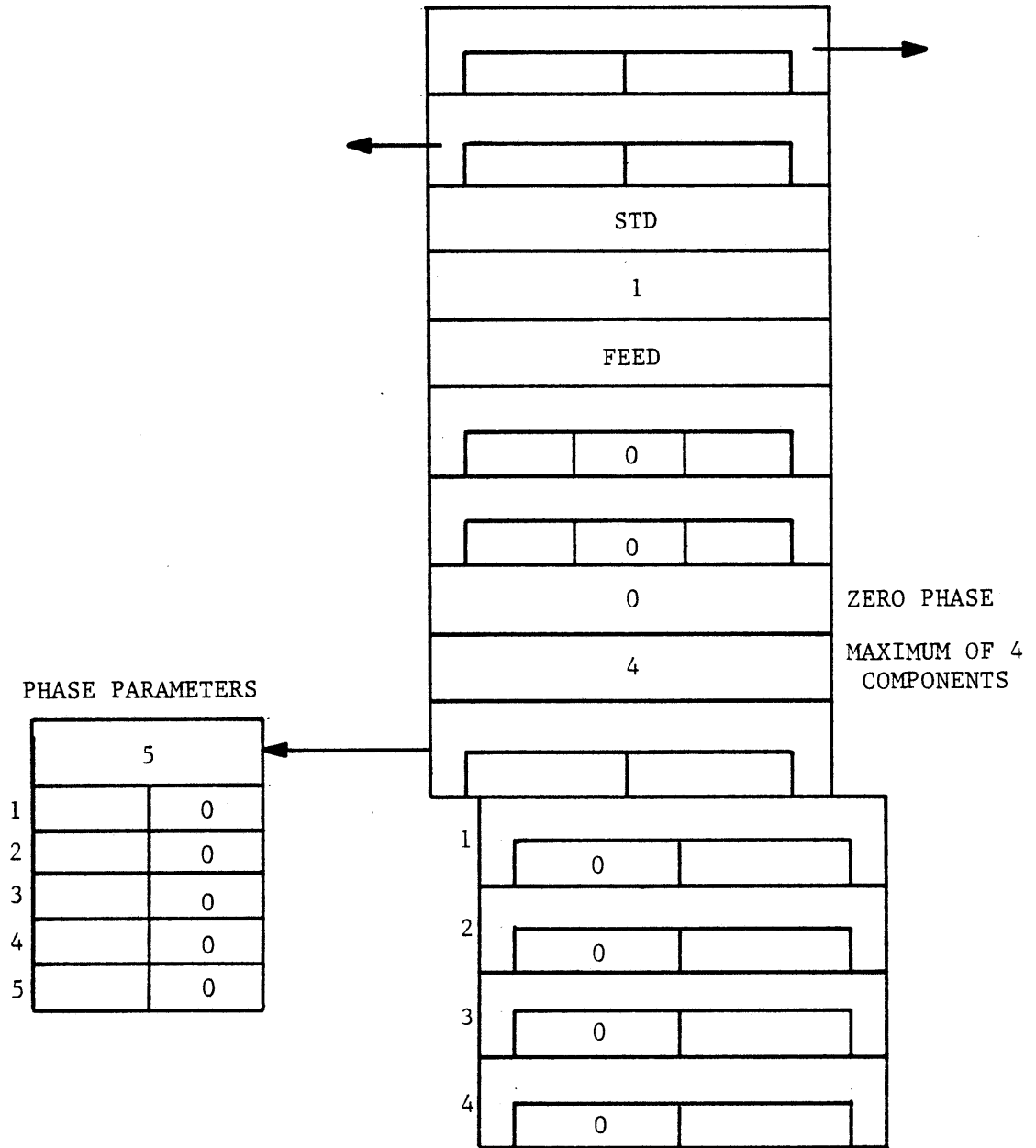


FIGURE 5.9a AN EXAMPLE OF A STREAM STRUCTURE WITH NO COMPONENTS

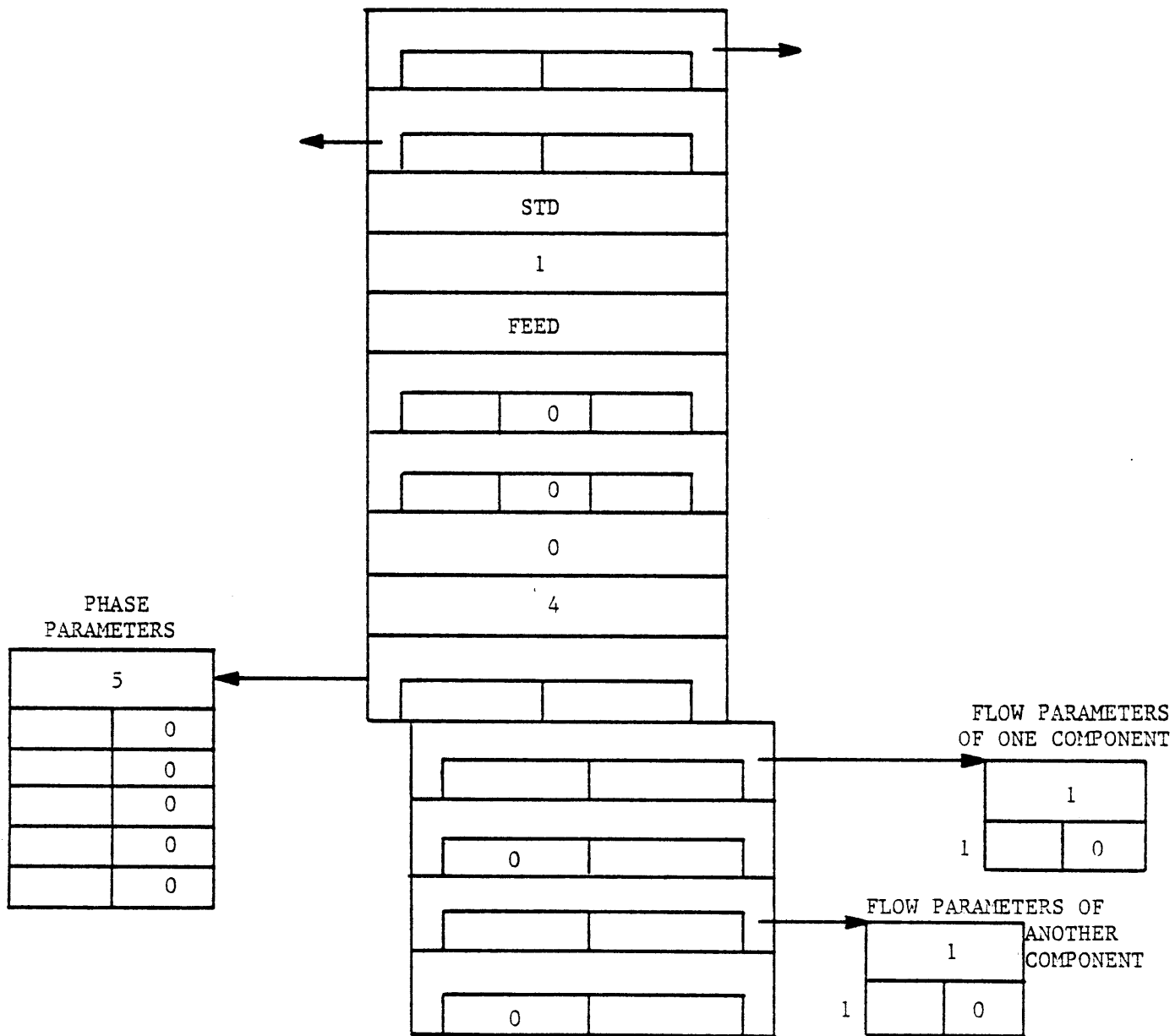


FIGURE 5.9b AN EXAMPLE OF A STREAM STRUCTURE WITH SOME COMPONENTS

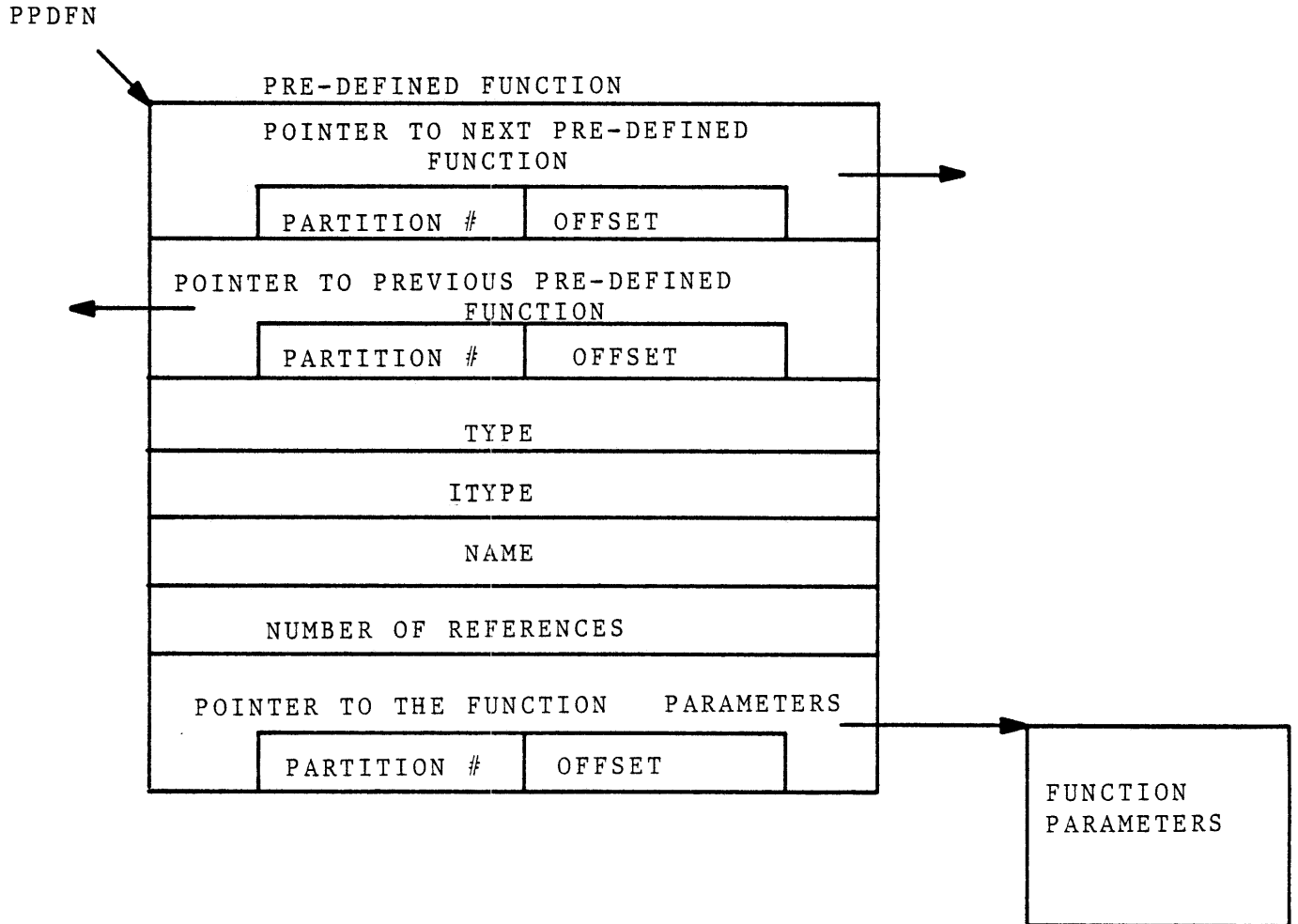


FIGURE 5.10 THE PREDEFINED FUNCTION STRUCTURE

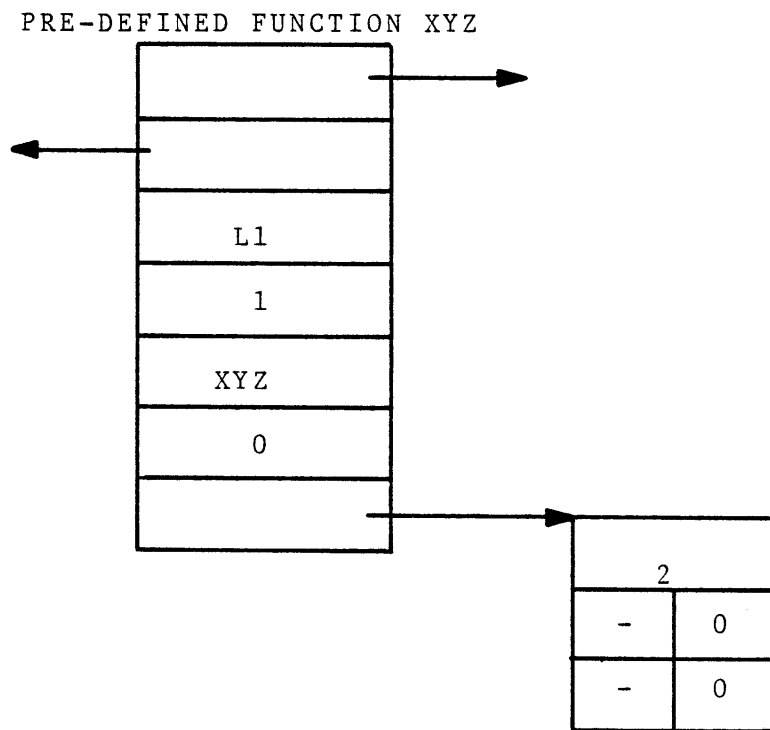


FIGURE 5.11 AN EXAMPLE OF A
PRE-DEFINED FUNCTION STRUCTURE

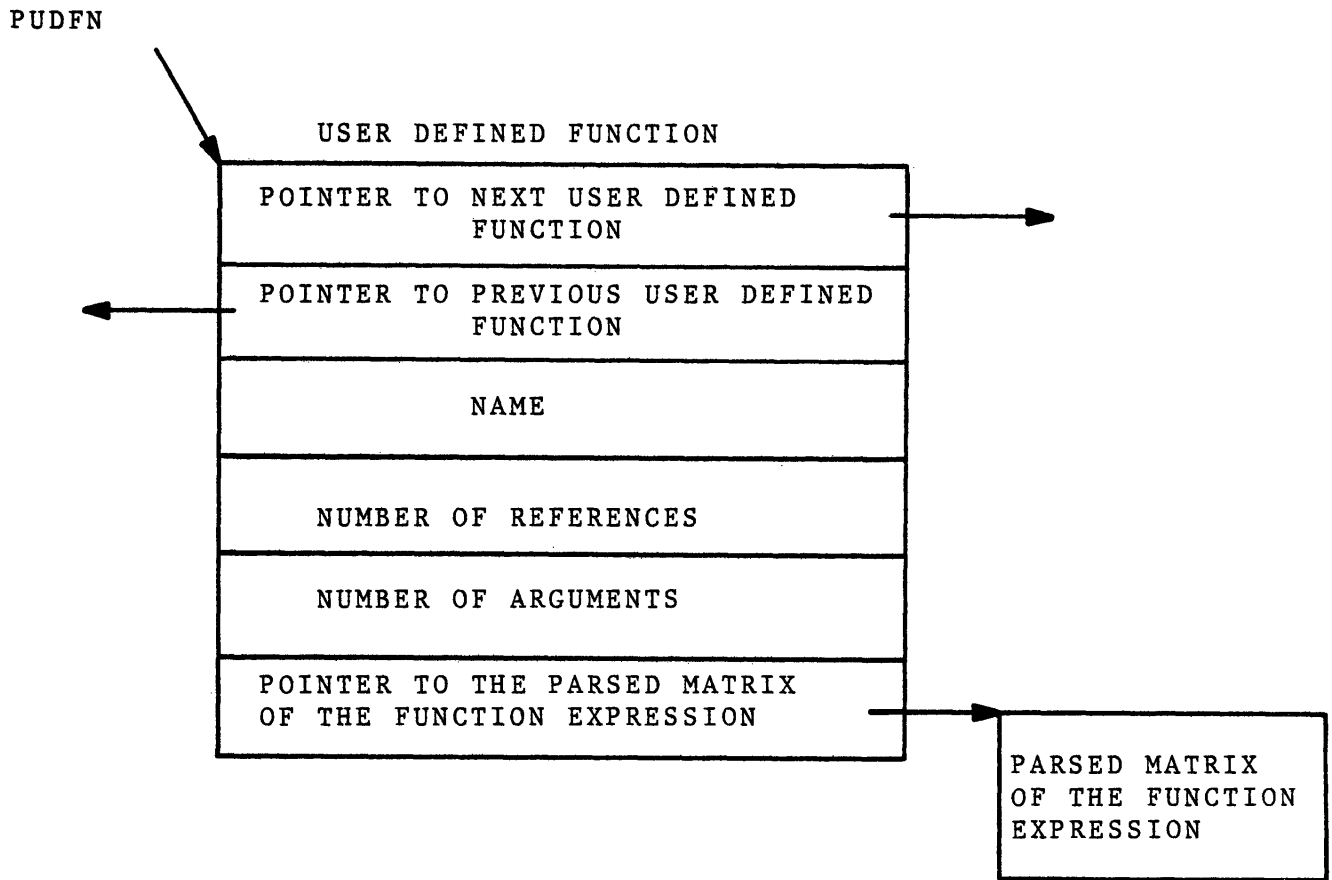


FIGURE 5.12 THE USER DEFINED FUNCTION STRUCTURE

- d) Number of references. The number of times the function is referred to by other existing user-defined functions. This is to protect the users from deleting a function which is referred to by other existing functions.
- e) Number of arguments.
- f) A pointer to a data structure which represents the arithmetic expression used for defining the function. This data structure is called a "Parsed Matrix". In this matrix, operations of the expression are listed sequentially in the order they would be executed to evaluate the expression. The parsed matrix data structure is also used to represent any arithmetic expression encountered in the user's commands as will be discussed in Chapter 9. The full description of the parsed matrix will be given in Chapter 9. Each matrix entry has one operator and two operands. Operators are represented by integer numbers. Each operand is represented by a set of three items, one of them a pointer pointing to the data structure representing the operand. For efficiency reasons, these pointers are absolute pointers, opposed to the type of pointers defined earlier in this chapter as a pair of partition number and offset locator. Hence the user-defined functions, unlike the other process elements (e.g., units, streams, etc.), are not created inside the working area and cannot be saved. They are only valid for the duration of the active process. The variable "udfn_head" points to the beginning of the user-defined function's list.

The data structure representing the function created by the following user command: `create function (y(X1, X2) = X1 + X2*4);` is shown in Figure 5.13. Suppose two other functions were earlier defined. For ease of reading the operators and operands of the matrix are shown as character strings. ARG1 and ARG2 represent the first and second arguments, respectively. M_i denotes the result of the i th matrix entry.

5.2.7 Variable Data Structure

The variable data structure is shown in Figure 5.14. It contains the following:

- a) A pointer to the next variable.
- b) A pointer to the previous variable.
- c) Variable's name.
- d) Variable's value.

For example, in response to the following user command:

`sp v (X=2*4,Z=12);` the data structures shown in Figure 5.15

will be added to the beginning of the variable list.

5.2.8 Data Structure Containing the Property Estimation Methods in Use

This data structure is shown in Figure 5.16. It contains the following:

- a) number of properties.
- b) An entry for each property containing the estimation method in effect.

When a new process is created, the entries of this data structure are initialized to the default options given in the template for property estimation methods table. Entries are updated in response to the user's "use" commands.

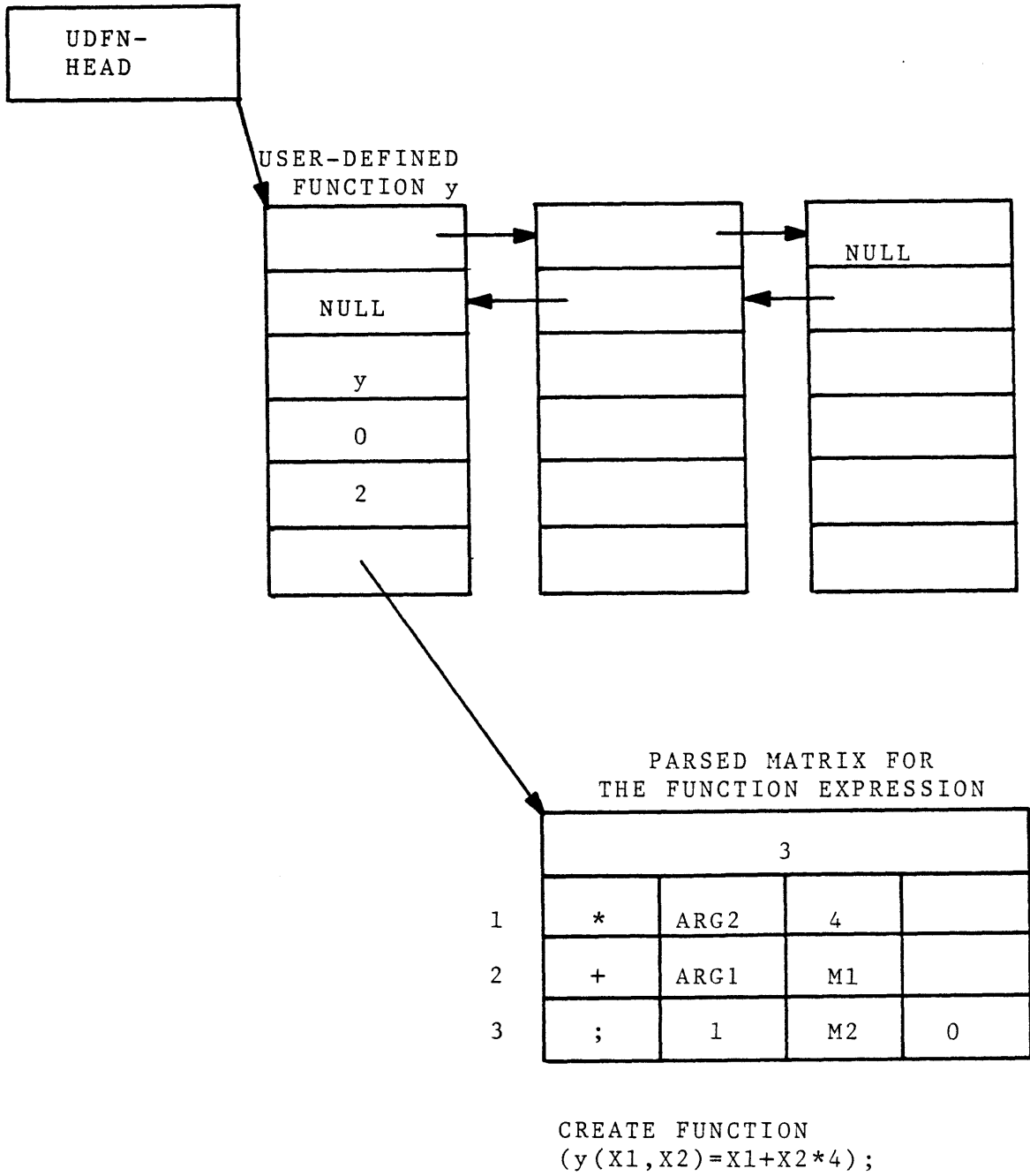


FIGURE 5.13 AN EXAMPLE OF A USER DEFINED FUNCTION STRUCTURE

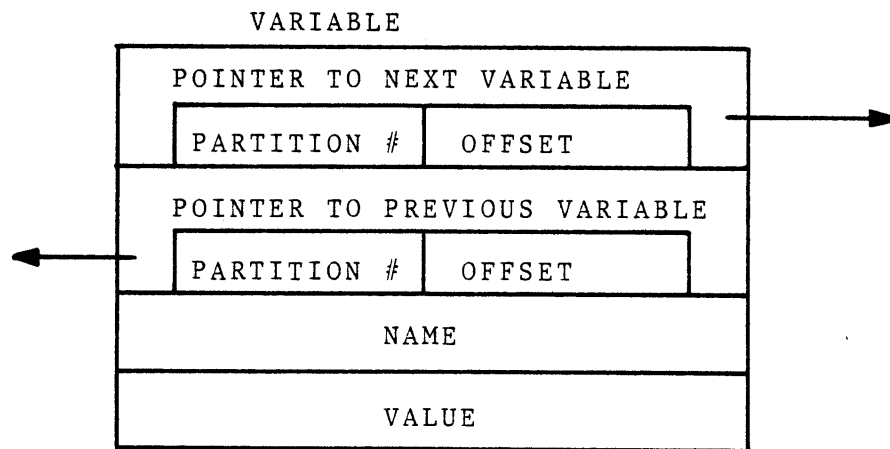


FIGURE 5.14 THE VARIABLE STRUCTURE

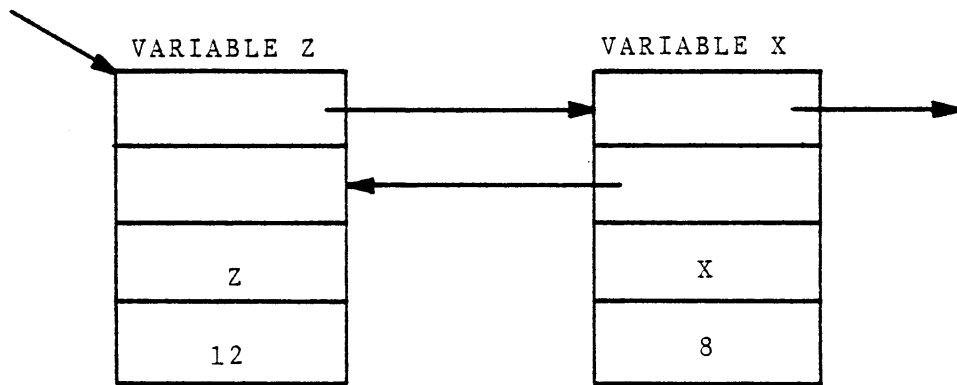


FIGURE 5.15 AN EXAMPLE OF A VARIABLE STRUCTURE

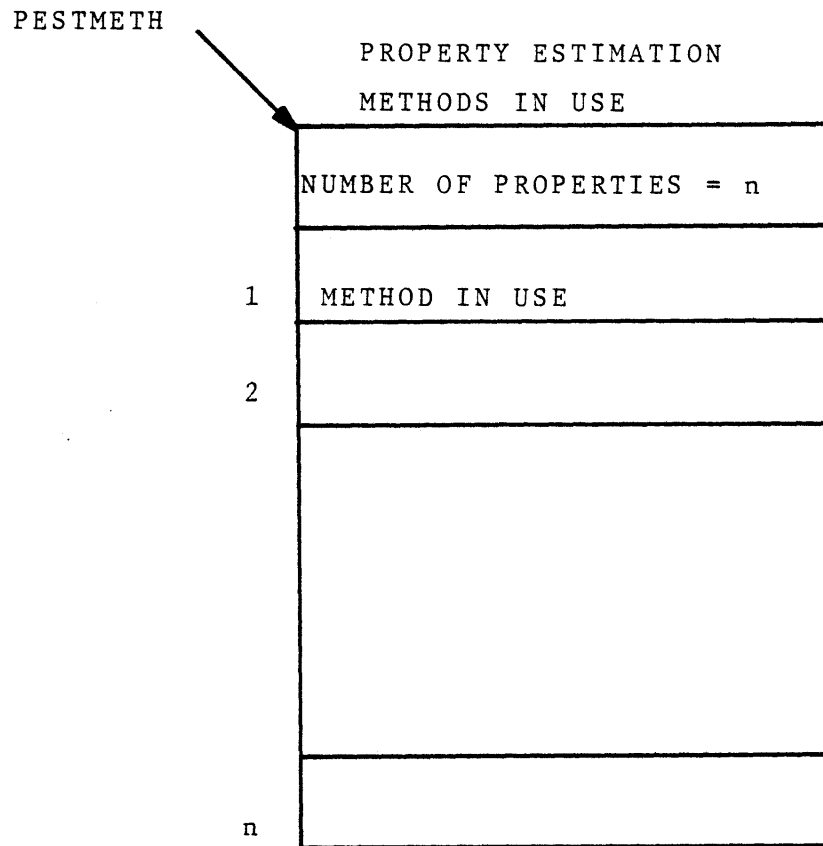


FIGURE 5.16 THE DATA STRUCTURE FOR PROPERTY ESTIMATION
METHODS IN USE

For example, given the template in Figure 4.14, the data structure shown in Figure 5.17a will be created. Figure 5.17b shows the data structure after the following command:

```
use FVAP=2 ALIQ=5;
```

Associated with each process there is such a data structure which is saved or retrieved along with other data structures constructing the process, when the process is saved or retrieved. The external pointer "pestmeth" contains the location of the data structure, enabling the physical property estimation routines to access the data structure for retrieving the specified methods of properties estimation.

The estimation routine will avoid requirement for this pointer by using the following service routine to retrieve the methods to be used:

```
call get_meth (property_number, method, error_code);
```

The description of service routines is given in Chapter 6 and Appendix C.

5.3 Process Files

A user may have any number of process files, each containing any number of processes. A process file has a directory and a number of processes. The process file directory resides on a single segment called "Xdr.GPES" where X is the process file's name. Every partition of each process resides on a separate segment, named XYn.GPES, where X is the process file's name, Y is the process name, and n is the partition number. All segments of a process file reside under one directory. The data structure of the directory for a process file is shown in Figure 5.18. It contains the name, serial number, and compatibility level of the system under which the directory has been created and the current number of entries. Each entry is either empty or represents a process.

	4
1	1
2	1
3	4
4	1

(A)

	4
1	1
2	2
3	4
4	5

(B)

FIGURE 5.17 AN EXAMPLE OF THE DATA STRUCTURE FOR PROPERTY ESTIMATION METHODS IN USE.

DIRECTORY OF A PROCESS FILE

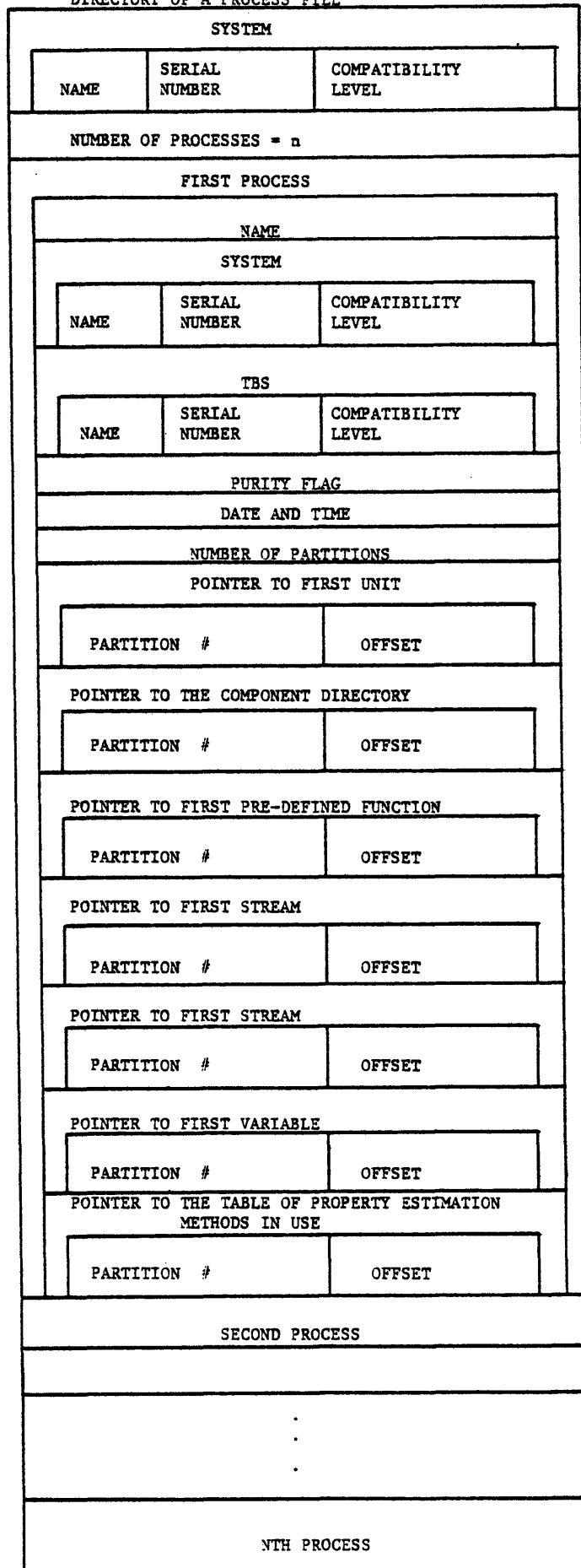


FIGURE 5.19 THE DIRECTORY OF A PROCESS FILE

Each entry contains the following:

- a) Process name. For an empty entry this field is blank.
- b) Name, serial number, and compatibility level of the system under which the process has been first created.
- c) Name, serial number, and compatibility level of the TBS under which the process has been first created.
- d) Purity flag. This flag is "off" if the process has ever been accessed by any incompatible system or TBS. It is "on", otherwise.
- e) Date and time when the process was first created.
- f) Number of partitions.
- g) A pointer to the beginning of unit structure list.
- h) A pointer to the component directory.
- i) A pointer to the beginning of the pre-defined function structure list.
- j) Two pointers to the beginning of the stream structure list.
- k) A pointer to the beginning of the variable structure list.
- l) A pointer to the data structure of the property estimation methods in use.

An example of a process file is shown in Figure 5.19.

5.4 Component Files

A user may have any number of component files, and each may contain data for any number of components of many component types. The users may share their component files. Users component files are called private component files. The TBS administrator may create such component files and make them available to all users. These files are called public component files. A TBS may have any number of public component files which may be

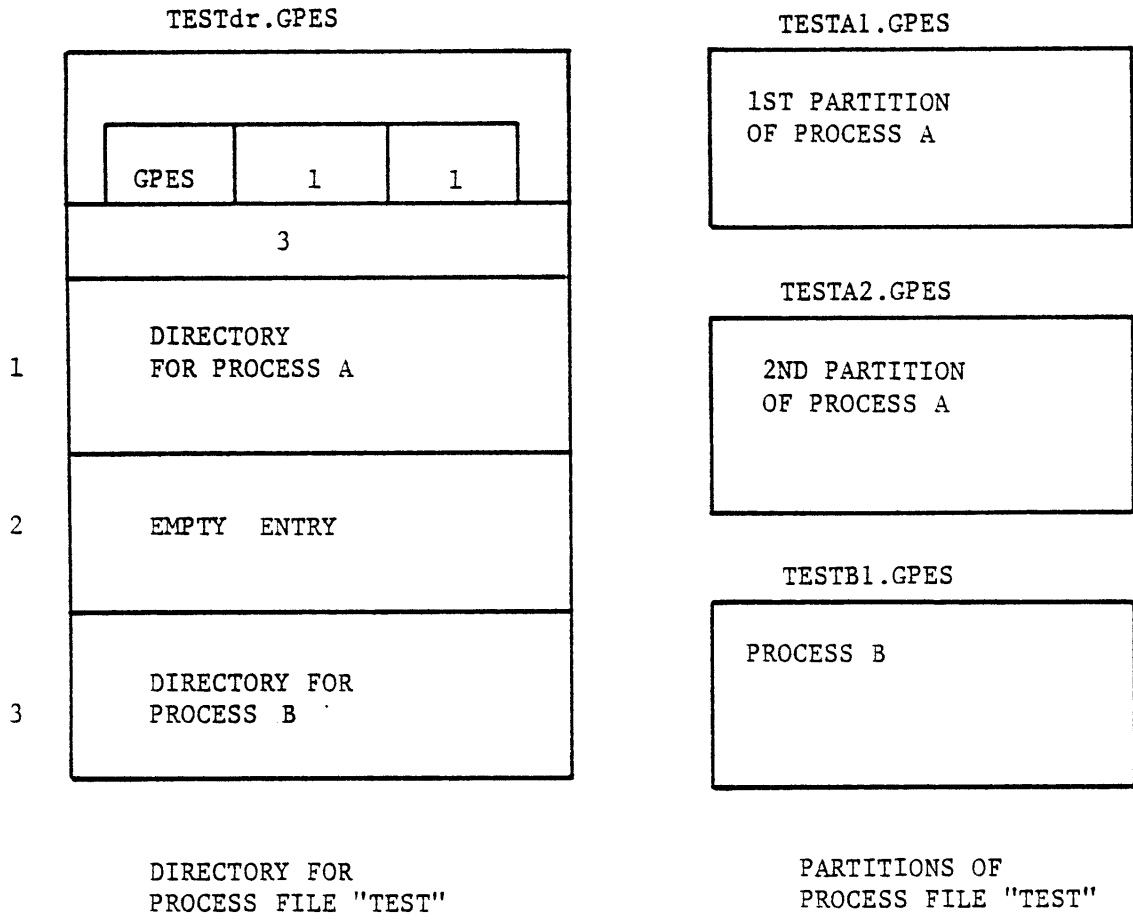


FIGURE 5.19 AN EXAMPLE OF A PROCESS FILE

available to all or some users of the TBS. The data structure representing a component file is shown in Figure 5.20. It contains the following:

- a) Name, serial number, and compatibility level of the system under which the file has been first created.
- b) Name, serial number, and compatibility level of the TBS under which the file has been first created.
- c) Purity flag indicating whether the file has or has not been accessed by any incompatible system or TBS.
- d) Remark, which may describe the source of data or references, etc.
- e) Maximum number of component types that the file may contain.
- f) The above number of entries. Each entry representing a component type, and contains the following: Component type, number of parameters for each component of that type, and a pointer to the first component of that type. For empty entries, component type is blank.
- g) A space in which all component data structures reside.

The structure of a component in the file is shown in Figure 5.21. It contains the following:

- a) A pointer to the next component in the list.
- b) A pointer to the previous component in the list.
- c) Component's name.
- d) Number of component parameters.
- e) The value of each parameter.

For saving in storage and efficiency in processing the value type of parameters are not stored. Therefore, the components having unspecified parameters cannot be saved.

COMPONENT FILE

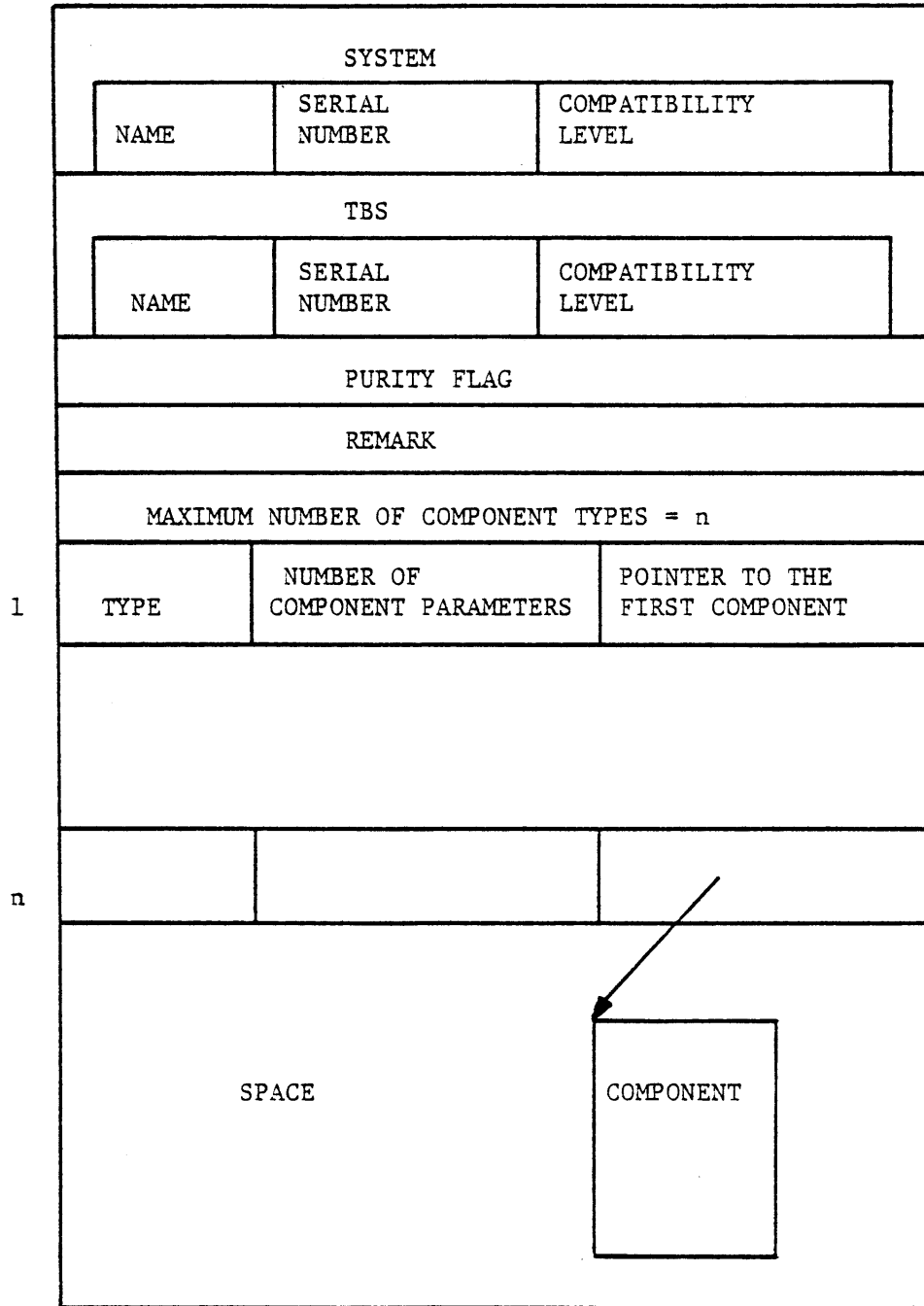


FIGURE 5.20 THE COMPONENT FILE STRUCTURE

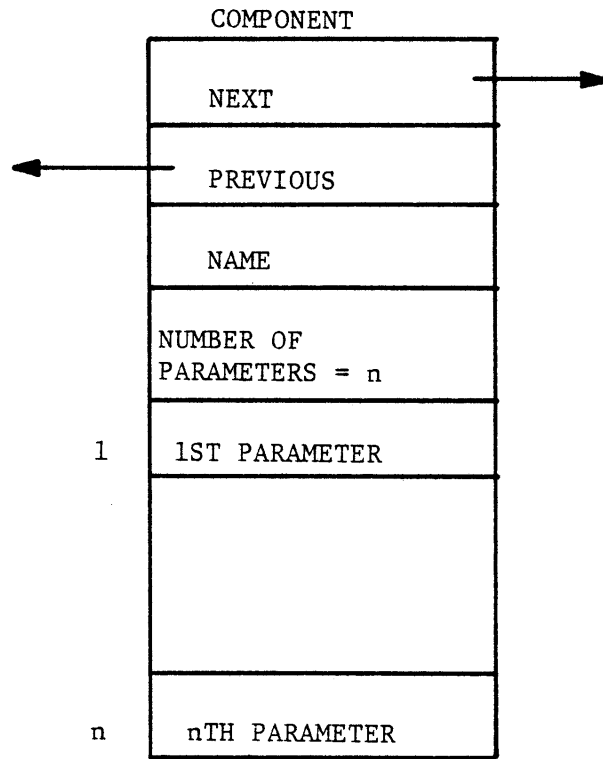


FIGURE 5.21 THE STRUCTURE OF A COMPONENT IN THE COMPONENT FILE

All components of each type are linked together by next and previous pointers. Components in the list are in alphabetical order. Each component file resides on a single segment, hence having a limited capacity of about 10^6 bytes. The segment name is Xc.GPES where X is the component file's name. An example of a component file is shown in Figure 5.22.

DATAc.GPES

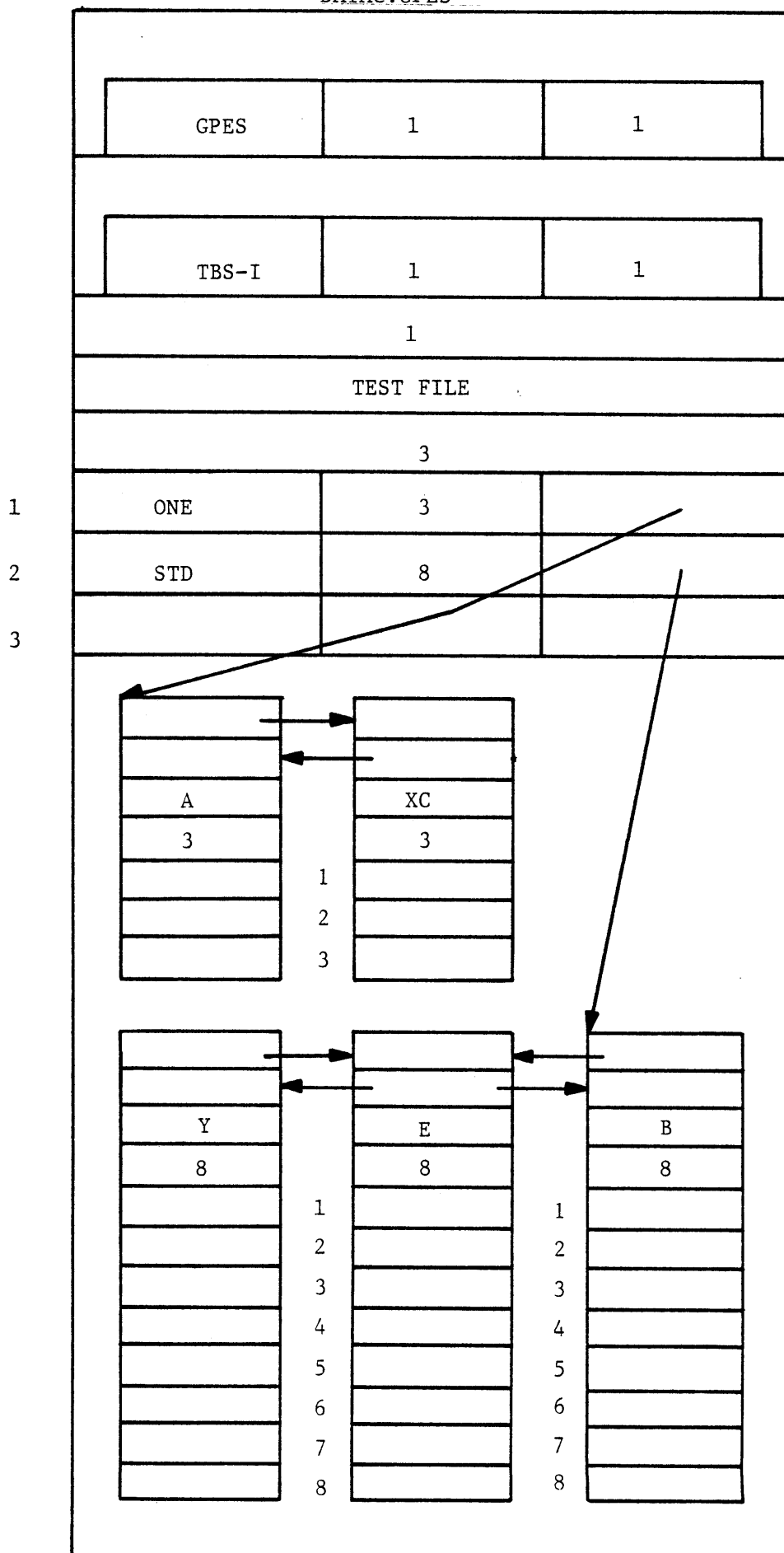


FIGURE 5.22
AN EXAMPLE OF A
COMPONENT FILE

CHAPTER 6TBS PROGRAMS

As discussed earlier a TBS consists of a set of definitions (templates) and a package of subroutines (programs). TBS programs are the heart of a TBS. TBS programs can be divided into two classes: Primary programs and secondary programs. Primary programs are those programs that are called directly by the GPES Executive. Secondary programs are those programs that are called by primary programs as shown in Figure 6.1.

6.1 Primary Programs

Primary programs are called by the system either in response to the user's calculate commands or in case of a pre-defined function for evaluating the value of the function. Hence primary programs can be further divided into two groups: calculating routines and pre-defined function evaluating routines. The templates of a TBS contain the names and other information regarding the primary programs.

6.1.1 Calculating Routines

Associated with each type of unit, stream, component, or function is usually a calculating routine. Unit calculating routines which are referred to in the literature as process modules, building blocks, etc. are the essential part of a TBS. In general they represent the functional relationship between inlet streams, unit parameters, and outlet streams. Stream and component calculating routines are not commonly used. But in general a stream calculating routine may calculate a number of phase or flow parameters as a function of the thermodynamic state variables and chemical composition of the phase or stream. Examples are calculating

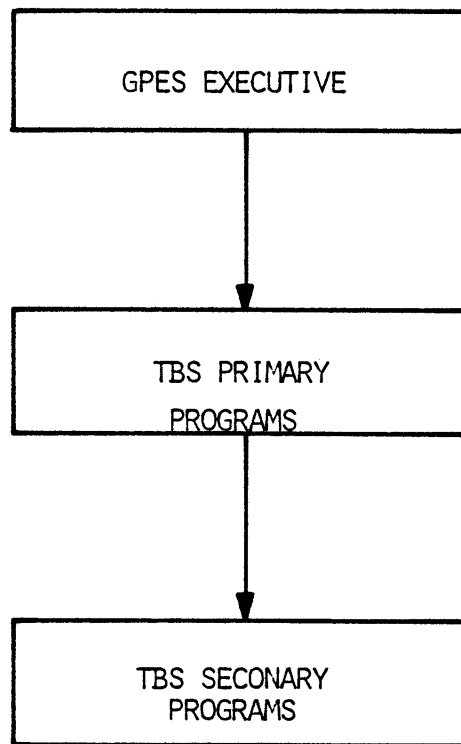


FIGURE 6.1 THE INTERACTION BETWEEN THE GPES EXECUTIVE AND TBS PROGRAMS.

enthalpy and viscosity. Similarly a component calculating routine may calculate a number of component parameters as a function of other parameters, and/or some experimental data.

A function calculating routine may perform any algebraic, transcendental, or integral operation or carry out such operations as regression analysis on input data and calculate the function parameters (e.g. regression coefficients).

It should be noted that not every unit, stream, component, or function type need to have a calculating routine. Using a ";" as the calculating routine's name in the template indicates that no calculating routine is specified with that template.

The GPES design has been based on the assumption that for simulating a process flowsheet the user of a TBS will provide the order in which individual process units be calculated in his calculate command. However, provision has also been made for implementing template based systems that could simulate the entire process flowsheet without requiring the user to provide the calculation order. Such a TBS should have a calculating routine to perform this task. Therefore, in general a TBS in addition to individual calculating routines for each unit type, stream type, etc. may also have a calculating routine for all units, a calculating routine for all streams, a calculating routine for all components, and/or a calculating routine for all functions. These calculating routines, if any, will be called in response to the following users commands:

```
calculate unit all (optional argument list);  
calculate stream all (optional argument list);  
calculate component all (optional argument list);  
calculate function all (optional argument list);
```

Therefore, calculating routines can be divided to the following two groups: Individual calculating routines and "all" calculating routines as shown in Figure 6.2.

Each group may also be subdivided to the following four subgroups: unit, stream, component, and function. Table 6.1 shows the number of programs in each group for a TBS.

6.1.2 Pre-defined function evaluating routines

For each pre-defined function in addition to a calculating routine there may be another routine known as an evaluating routine. When a pre-defined function appears in an arithmetic expression in the user's command, such a routine is called by the GPES executive. Its function is to evaluate the function given the function parameters (coefficients) and the function arguments.

6.2 Secondary Routines

Secondary programs are those routines that are called by primary routines. Physical property estimation routines fall into this class of routines. System-supplied service routines that are called by other routines to perform various tasks also fall into this class of routines. Service routines are described later.

6.3 Interaction Between TBS Programs and the GPES Executive

In a Fortran environment, in most chemical process simulation systems, the process module is written as a subroutine with the transfer of variables taking place through the parameter list of a call statement or through common blocks of data. In GPES environment, the process of transferring variables is more elaborate. The TBS program receives the pointers to the data structures which contain the values of the variables. Thus, before beginning the calculations, the program must retrieve these

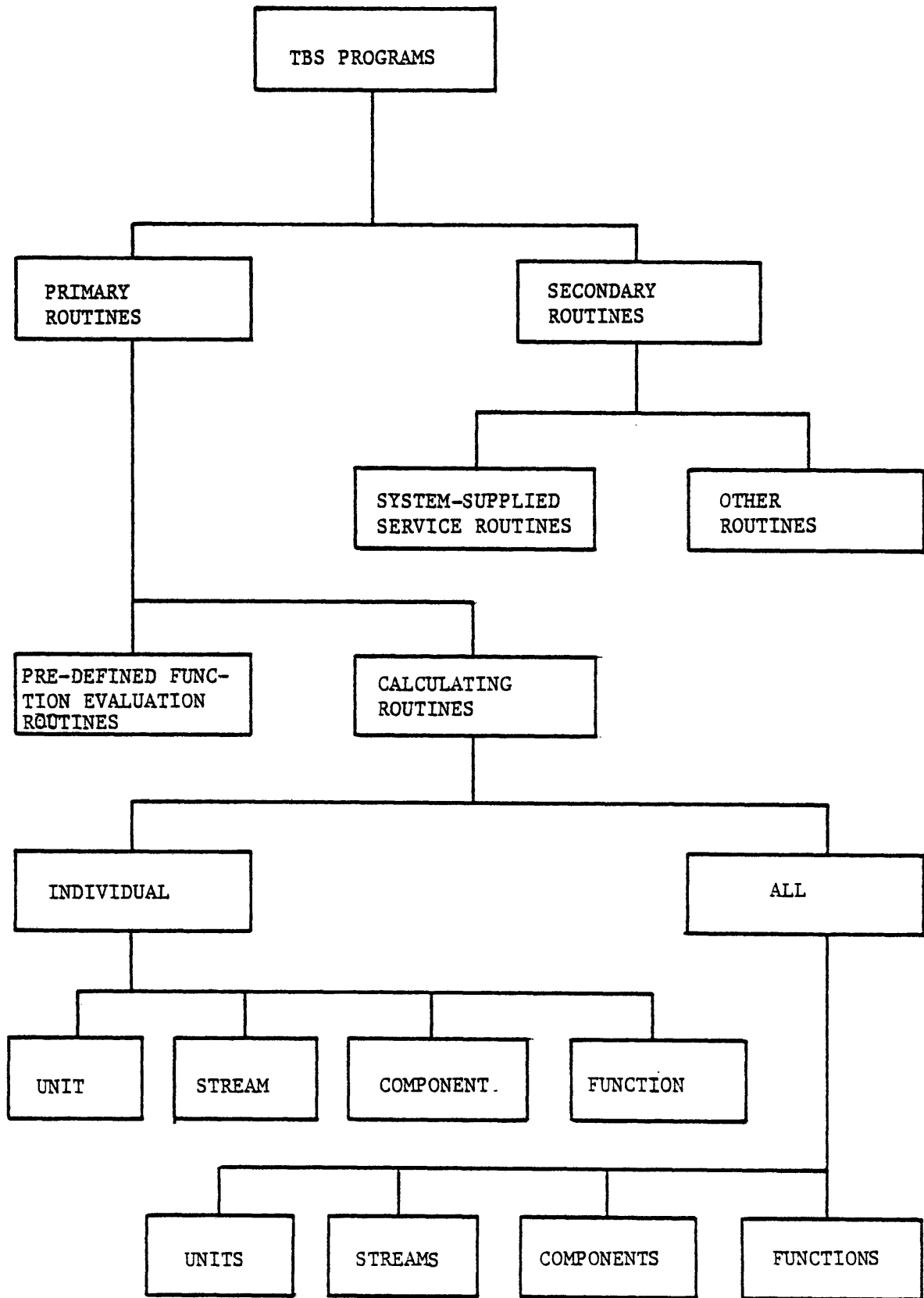


FIGURE 6.2 CLASSIFICATION OF TBS PROGRAMS

TABLE 6.1 Number of Primary Programs in a TBS

<u>Primary Programs</u>	<u>Number</u>
Calculate Individual	
Unit:	one or none for each unit type
Stream:	one or none for each stream type
Component:	one or none for each component type
Function:	one or none for each pre-defined function type
Calculate All	
Units:	one or none for the TBS
Streams:	one or none for the TBS
Components:	one or none for the TBS
Functions:	one or none for the TBS
Evaluate	
Function:	one or none for each pre-defined function type

values from the data structures. This is done by reaching the variable location through a series of pointers. The same must be done at the end of the calculations to store the results into the data structures network. This process of reaching the variable location in the network of data structures is called "navigation". The TBS program calls upon some system-supplied service routines to perform this process of navigation. The service routines will be described later. This method of variable transfer enables GPES to call every TBS program by a few generalized call statements with standardized parameter lists, using the powerful means of dynamic linking.

GPES will, in effect, call each group of TBS programs by the call statements shown in Table 6.2.

The interaction between the GPES Executive and a unit calculating routine (process module) is shown in Figure 6.4. Unit computations, which are the heart of a unit calculating routine, can either be performed by the calculating procedure or by a call on an already developed FORTRAN or PL/1 program (a secondary program).

Although a TBS program usually has access to all relevant information through the pointers passed to it, some TBS programs (especially "all" calculating routines) may require additional information which might not be accessible through those pointers. A number of external variables have been provided which enables those special routines to access any information about the process flowsheet. These external variables are listed in Table 6.3.

Table 6.2Call Statements used by GPES Executive to Invoke a TBS Primary Program.

<u>Group of Programs</u>	<u>Call Statement</u>
Calculate Individual	
Unit:	call routine (punit,parg,switch,error_switch);
Stream:	call routine (pstream,parg,switch,error_switch);
Component:	call routine (comp_index,parg,switch,error_switch);
Function:	call routine (pfunc,parg,switch,error_switch);

Calculate All

Units:	call routine (parg,error_switch);
Streams:	call routine (parg,error_switch),
Component:	call routine (parg,error_switch);
Functions:	call routine (parg,error_switch);

Evaluate a Function: call routine (pparm,arg,result)

Where:

routine is the name of the appropriate routine.

punit is the pointer to the unit structure (input).

pstream is the pointer to the stream structure (input).

comp_index is the index (fixed bin(17)) of the component in the component directory (input).

pfunc is the pointer to the pre-defined function structure (input).

parg is the pointer to the arguments structure (input).

The arguments structure as shown in Figure 6.3 contains the arguments the user provides in the calculate command.

Table 6.2 Contined

The first argument is always interpreted as the level of calculation. The interpretation of other arguments, if any, should be clear both to the user and to the calculating routine. If no argument is provided (parg = null) the level of calculation is assumed to be one.

<code>pparm</code>	is the pointer to the function's parameters structure (input).
<code>switch</code>	(bit(1)) Indicates whether convergence has been achieved (output). It is only used for convergence routines which are described in section 6.4.4
<code>error_switch</code>	(bit(1)) Indicates whether any severe error has been detected by the routine (output).
<code>arg</code>	(dim(*) float bin (63)) is the array of function arguments (input).
<code>result</code>	(float bin (63)) is the result of the function evaluation (output).

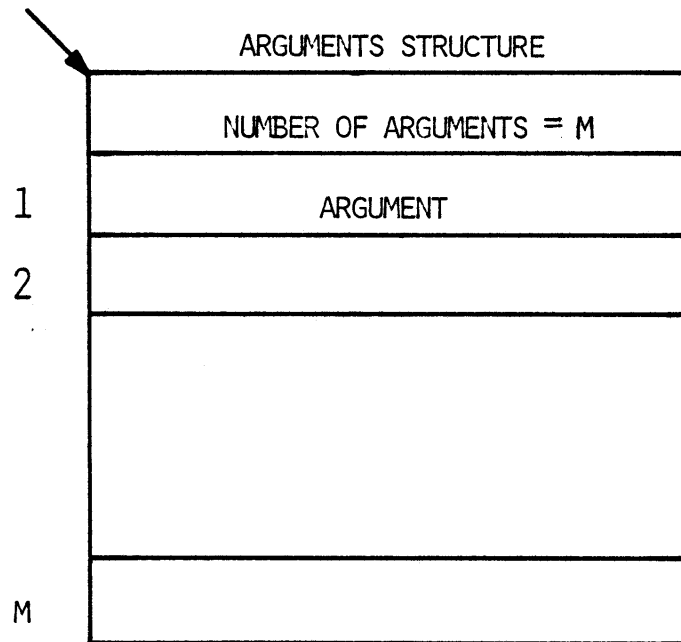


FIGURE 6.3 THE ARGUMENTS STRUCTURE

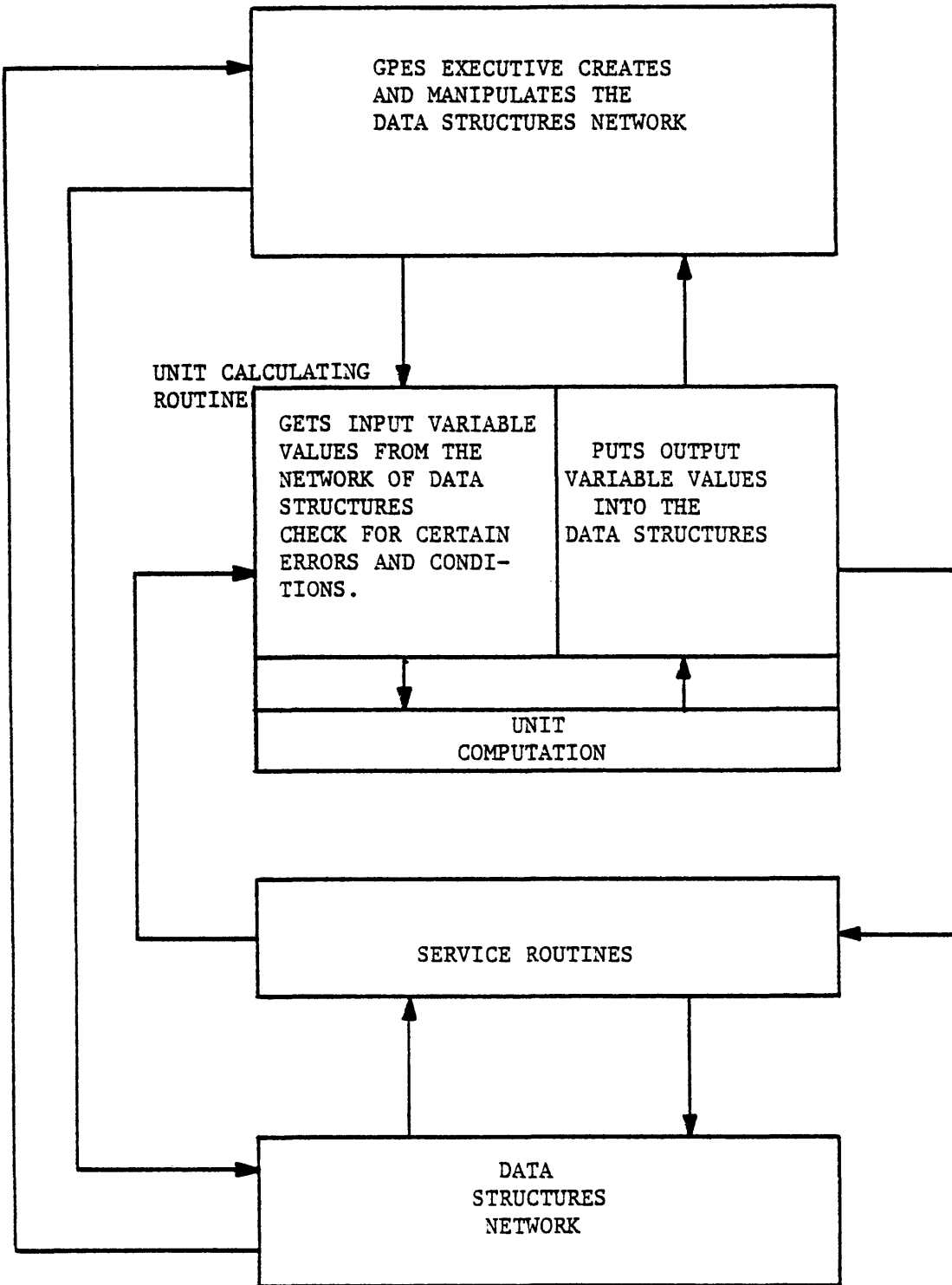


FIGURE 6.4 INTERACTION BETWEEN GPES EXECUTIVE, A UNIT CALCULATING ROUTINE, AND SERVICE ROUTINES

Table 6.3

External Variables For Use By TBS Programs

<u>Variable</u>	<u>Description</u>
p_dr	A pointer to the process directory data structure (shown in Figure 5.1). Used by "all" calculating routines.
pcomp_dr	A pointer to the component directory (shown in Figure 5.6). Used by component calculating routines and physical property estimation routines. By using the service routines there is no need for this variable.
pestmeth	A pointer to the data structure of "property estimation methods in use" (shown in Figure 5.16), used by property estimation routines. If the service routine "get_meth" is used there is no need for this variable.
output_file	A file variable indicating the output file. It represents either the terminal or another file. TBS programs should write error messages on the terminal. Other outputs (if any) are recommended to be printed on the output file specified by this variable.

Table 6.3 Continued

<u>Variable</u>	<u>Description</u>
s	It is an integer number (fixed bin (17)) indicating the significant number of digits to be printed for numerical data. A TBS program may or may not use this number for its printout (if any).
d	It is an integer number (fixed bin (17)) indicating the decimal number of digits to be printed for a numerical data. A TBS program generating some output may or may not use this number.

6.4 Writing a TBS Program

In this section some general considerations on writing a TBS program are discussed. Examples in writing TBS programs is given in Chapter 10.

6.4.1 Input

Usually all the input data required by a TBS program are stored in the process network. To retrieve these data, the TBS program may call upon some service routines which in effect navigate through the process network. Nevertheless a TBS program may also interact directly with the user for:

- a) getting additional data, or
- b) asking user's preference where there is more than one alternative available to the routine. In this case the TBS program should write a message on the user's terminal requesting such data and then call upon some other service routines for receiving the data.

6.4.2 Output

Usually the TBS program will store the computed data into the process network by calling upon the appropriate service routines. Nevertheless, the TBS program may also write the intermediate or final results onto the user's terminal or user's output file. The user's output file is either the user's terminal or another file. The external file variable "output_file" indicates the user's output file. Therefore, a TBS program writing on this file should:

- a) provide the following declaration:

```
declare output_file file variable external;
```

- b) The "put" statements should specify the file as follows:

```
put file (output_file)...;
```

6.4.3 Error Detection

Before the GPES executive passes control to a calculating routine for a specific level of calculation, it performs the various types of error checking requested in the associated template for that level of calculation. This means that a calculating routine should not be concerned about over- and under-specification of parameter values. In other words, input data are provided (i.e., specified, assumed, or calculated), output data are not fixed (i.e. not specified), and initial value assumptions (if any) are provided and not fixed (i.e. assumed, or calculated). In the case of a unit calculating routine, the necessary inlet and outlet streams are also connected to the unit. In addition to the above types of error checking which are performed by the GPES executive, there are numerous other types of error checking which a TBS program may perform. The extent of error checking that a program may undertake depends on the generality of the program and the assumption about the user's knowledge of the program and his common sense. The following are some examples of types of error checking that a TBS program may perform:

- a) unacceptable input data (e.g. negative numbers for flow rates, mole fractions outside the range of 0 to 1, sum of mole fractions more than one).
- b) an internal algorithm fails to function (e.g. for a trial-and-error calculation, convergence can not be achieved).
- c) violating general laws of thermodynamics.
- d) in the case of a unit calculating routine, if the unit is such that only certain components are allowable in an inlet or outlet stream, or if the components of one stream are required to be identical as those in another stream, the routine should also

check for these conditions. Both of these checking procedures can be accomplished easily using service routines.

Once a TBS program detects a serious error it should perform the following:

- a) write on the user's terminal a description of the detected error.
- b) turn "on" the error switch: `error_switch = "1"` b;
- c) either return immediately to the calling program or continue further error detection. In the latter case the program should immediately return to the calling program after the error detection task is completed.

The error detection task should be accomplished prior to writing the computed results into the process network. Therefore, it is recommended that the transferring of the computed results into the process network be performed as the last task of the routine.

6.4.4 Convergence Routines

Most chemical processes involve recycle streams. Therefore, such flowsheets contain information recycle loops, that is, cycles for which too few stream variables are known to permit equations for each unit to be solved independently. The equations for units in an information recycle loop must be solved simultaneously. One solution technique is to "tear" one stream in the recycle loop [11,22,51,148] that is, to guess variables of that stream. Based upon tear stream guesses, information is passed from unit-to-unit until new values of the variables of the tear stream are computed. These new values are used to repeat the calculations until convergence tolerances are satisfied. A TBS capable of analyzing flowsheets with recycle streams should have one or more types of pseudo units for testing and promoting convergence. This type of pseudo unit is

known as a convergence unit or convergence block. To define a unit of type convergence, as in the definition of any other unit type, the TBS administrator should create a template in the template data base and should provide a calculating routine for it. Such a unit usually has one inlet stream (containing the newly computed variables of the tear stream), one outlet stream (containing the guess values of the tear stream) and a number of parameters (convergence tolerances, etc.). Of course, such a unit is not limited to one inlet and one outlet stream, but may have any number of either. The calculating routine for such a unit is known as a convergence routine. Such a routine should compare newly computed variables with guess values, and compute new guess values when convergence tolerances are not satisfied. In particular, it should perform the following:

- a) Based on some pre-established criteria it should perform a convergence test.
- b) If convergence has been achieved turn "on" the "switch" argument. This is the only required use of the "switch" argument which is one of the arguments passed to each calculating routine.
- c) If convergence has not been achieved it should estimate the parameters of the output streams.

The information regarding the criteria for convergence could be either part of a unit's parameters (this is usually the case), or part of the parameters of a stream to be tested, or it could be provided by some information streams input to the unit. The convergence acceleration methods most commonly used in existing simulators are the following:

- a) successive substitution,
- b) bounded Wegstein,
- c) dominant eigenvalue, and
- d) quasi-Newton.

6.4.5 Writing an "all" Calculating Routine

Although writing an individual calculating routine is a very simple and systematic task, writing an "all" calculating routine may not be quite so simple. An "all" calculating routine should determine the order of calculation and then monitor the execution of subsequent individual calculating routines. This is a rather straightforward task when writing an "all" calculating routine for streams, components, or functions, where there is no interaction between individual elements. However, a unit "all" calculating routine is more sophisticated, and such a routine should perform the following:

- a) Recognize the recycle streams.
- b) Determine the loops and the order of calculation in each loop.
- c) Either estimate the parameters of recycle streams or prompt the user for this information.
- d) Monitor the execution of individual calculating routines, and test for convergence.

The use of the "calculate unit all;" command and consequently the development of unit "all" calculating routines is not recommended for the following reasons:

- a) These routines require reasonable skill to develop.
- b) The same objective; i.e., calculation of the entire process can be achieved by having the user specify the order of calculation in the calculate command.
- c) The user may know something about the process that the unit "all" calculating routine may not know. Hence, letting the user decide on the order of calculation may result in faster convergence.

6.5 Service Routines

Service routines are developed to assist TBS programmers in conducting various tasks, of which the most important is navigation of the network of data structures. Although a TBS programmer using these service routines in his program may not be concerned about the details of the data structures representing process units, streams, or components, an understanding of the existence of such data structures and their interrelation will assure him of the proper usage of the service routines. Although there are about 100 service routines, usually only a few of these will be needed in writing a TBS program. Service routines perform various tasks, and can be divided into the following six groups:

- 1) Routines performing basic operations: Given a pointer to a data structure these routines return pointers to related data structures.
- 2) Routines checking for the existence of some components in a given stream.
- 3) Routines retrieving or storing the values of parameters from or into the process network. 78 out of 101 existing service routines fall in this group. Although using only two of them (`get_parm`, and `put_parm`) in conjunction with the basic service routines would be sufficient for accomplishing the above task, the other 76 routines have been provided for greater user (TBS programmer) flexibility and convenience. These routines can be divided into several subgroups as shown in Figure 6.5. They are used either for retrieving the values of parameters from the data structures (i.e., `get_parm`) or for storing the values of parameters into the data structures (i.e., `put_parm`). The reference to parameters may be

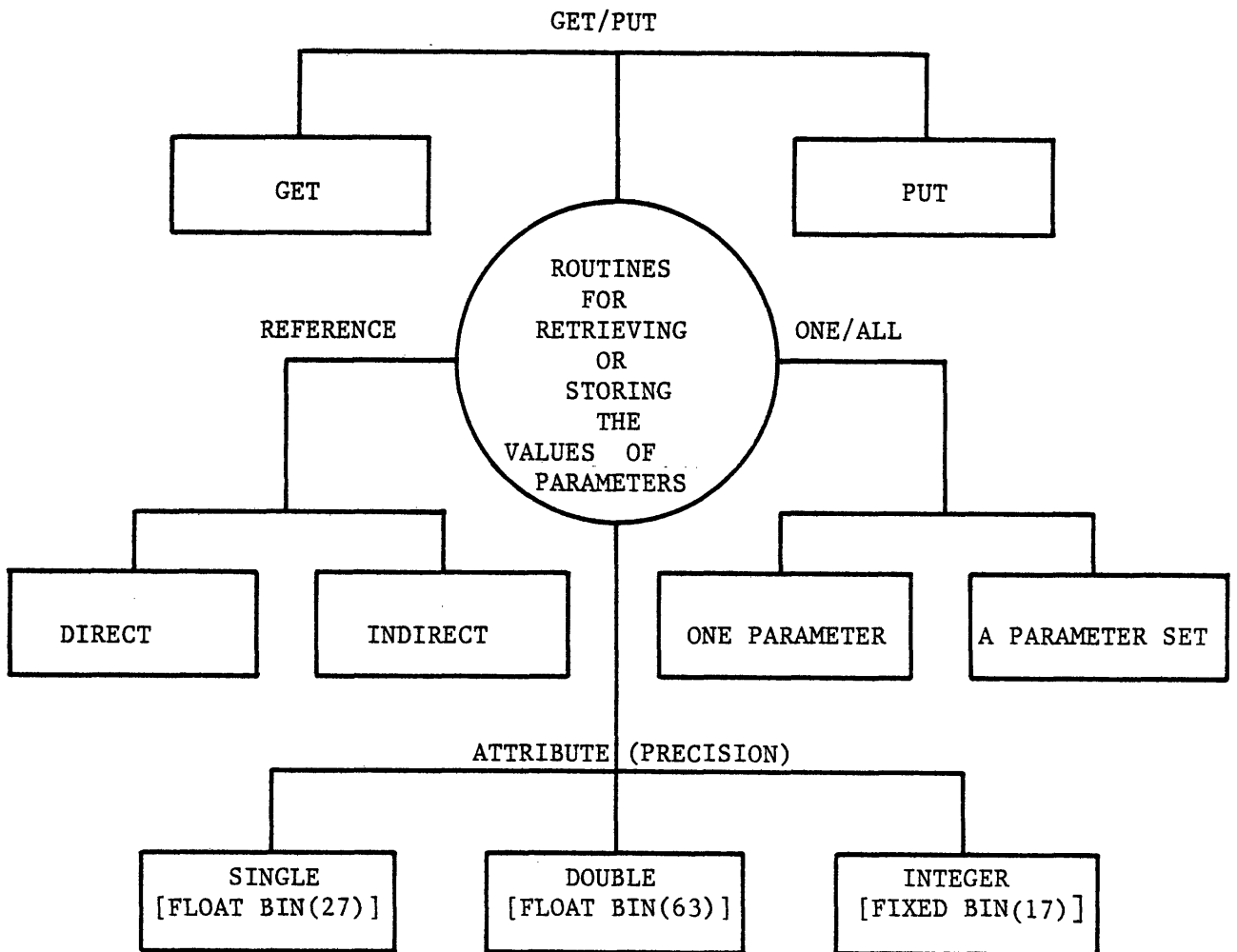


FIGURE 6.5 CLASSIFICATION OF SERVICE ROUTINES
RETRIEVING OR STORING THE VALUES OF PARAMETERS

made by a pointer to the parameter structure (direct), or through a pointer to a parent structure (indirect). For example, given "pparm", the pointer to a unit parameter structure, the call statement:

```
call get_parm(pparm,2,value,value_type, code);
```

will retrieve the value and value_type of the second unit parameter. The same result can be achieved by the call statement: call get_uparm (punit,2,value,value_type, code); where punit is the pointer to the unit data structure. Service routines get_parm and get_uparm are examples of direct and indirect service routines respectively. The service routines may retrieve or store either a single parameter (i.e. the above example) or all the parameters in a set. Routines get_parms and get_uparms are examples of the latter. Although the values of all parameters in all data structures are stored as double precision (FORTRAN terminology), in a TBS program the values of parameters may be supplied with other attributes. Service routines which provide different attributes to numerical data fall into one of three groups: those routines considering the values of parameters as double precision (i.e., get_parm), those considering the values of parameters as single precision (i.e., xget_parm), and those considering the values of parameters as integer numbers (i.e., iget_parm).

A list of service routines retrieving or storing the values of parameters is shown in Table 6.4.

- 4) Routines retrieving other variables of interest. Two routines fall in this group: one returns the values of user-supplied arguments, and another returns the method currently being used to estimate a physical property.

- 5) Routines directly interacting with the user. These routines are called to receive input data not provided in the process network, directly from the user.
- 6) Routines performing arithmetic operations on two parameter sets. These routines are helpful in writing process module calculations for separation processes and material balancing in general.

Table 6.5 contains a short description of each service routine. More detailed descriptions are given in Appendix C. A TBS administrator may provide TBS programmers with additional service routines especially developed for their particular needs and applications.

Table 6.5 A Short Description of Each Service Routine

<u>Group</u>	<u>Description</u>
1) <u>Basic Operations</u>	
spointer	Given a partition number and an offset it returns a pointer.
unit_ptr	Given the unit pointer, it returns the pointer either to the unit parameters or to the stream connected to the unit.
strm_ptr	Given the stream pointer, it returns the pointer to the parameters structure of a given phase.
get_comp_index	It returns the component index (entry number in the component directory) of a given component.
comp_ptr	Given the component index it returns the pointer to the component parameters.
func_ptr	Given the function pointer, it returns the pointer to the function parameters.
flow_ptr	It returns the pointer to the flow parameters of a given component in a given phase of a stream.
2) <u>Comparison Routines</u>	
same_comps	Determines if the components in one stream are those that are in another stream.
check_comps	Determines if the components present in a stream are those whose names are specified.

Table 6.5 Continued

<u>Group</u>	<u>Description</u>
3) <u>Retrieving or Storing the Values of Parameters</u>	
get_parm,put_parm, xget_parm,xput_parm, iget_parm,iput_parm	Given the pointer to the parameters structure each routine returns/stores the value of a given parameter.
get_parms,put_parms, xget_parms,xput_parms, iget_parms,iput_parms	Given the pointer to a parameters structure each routine returns/stores the values of all parameters.
get_uparm,put_uparm, xget_uparm,xput_uparm, iget_uparm,iput_uparm,	Each routine returns/stores the value of a given parameter of a unit.
get_uparms,put_uparms, xget_uparms,xput_uparms, iget_uparms,iput_uparms	Each routine returns/stores the values of all parameters of a given unit.
get_sparm,put_sparm, xget_sparm,xput_sparm, iget_sparm,iput_sparm	Each routine returns/stores the value of a phase parameter of a phase of a given stream.

Table 6.5 Continued

<u>Service Routine</u>	<u>Description</u>
get_sparms, put_sparms, xget_sparms, xput_sparms, iget_sparms, iput_sparms	Each routine returns/stores the values of all phase parameters of a given phase of a stream.
get_cparm, put_cparm, xget_cparm, xput_cparm, iget_cparm, iput_cparm	Each routine returns/stores the value of a parameter of a given component.
get_cparms, put_cparms, xget_cparms, xput_cparms, iget_cparms, iput_cparms	Each routine returns/stores the values of all parameters of a given component.
get_fnparm, put_fnparm, xget_fnparm, xput_fnparm, iget_fnparm, iput_fnparm	Each routine returns/stores the value of a parameter of a given function.
get_fnparms, put_fnparms, xget_fnparms, xput_fnparms, iget_fnparms, iput_fnparms	Each routine returns/stores the values of all parameters of a given function.
get_fparm, put_fparm, xget_fparm, xput_fparm, iget_fparm, iput_fparm	Each routine returns/stores the value of a given flow parameter of a component in a phase of a stream.

Table 6.5 Continued

<u>Service Routine</u>	<u>Description</u>
get_fparms, put_fparms, xget_fparms, xput_fparms, iget_fparms, iput_fparms	Each routine returns/stores the values of all flow parameters of a component in a phase of a stream.
get_fparamacs, put_fparamacs, xget_fparamacs, xput_fparamacs, iget_fparamacs, iput_fparamacs	Each routine returns/stores the values of a given flow parameter of all components, in a phase of a stream.
4) <u>Retrieving Other Variables of Interest</u>	
get_arg	It returns the value of a user-supplied argument.
get_meth	It returns the method currently being used to estimate a physical property.
5) <u>Interacting with the User</u>	
getrn	To receive a real number from the user.
getin	To receive an integer number from the user.
getline	To receive a line of characters from the user.
get_response	To receive a yes or no answer from the user.

Table 6.5 Continued

<u>Service Routine</u>	<u>Description</u>
<u>6) Arithmetic Operations on Two Parameter Sets</u>	
parms_operator1	This routine multiplies each parameter in a set of parameters by a given constant and adds to it another given constant. The results are stored in a data structure of comparable size corresponding to another parameter set: $p1_i = P2_i A + B$ for all i .
parms_operator1v	$P1_i = P2_i A_i + B_i$ for all i .
parms_operator2	$P1_i = P1_i + P2_i A + B$ for all i .
parms_operator2v	$P1_i = P1_i + P2_i A_i + B_i$ for all i .
fparmacs_operator1	It assigns a value to a given flow parameter for each component in a given phase of a stream equal to the product of a constant and another given flow parameter of the same component in a phase of another stream and added to another constant: $F1_i = F2_i A + B$ for all components. Where $F1_i$ and $F2_i$ are flow parameters of component i , but $F1_i$ may be in a different phase and stream than $F2_i$.
fparmacs_operator1v	$F1_i = F2_i A_i + B_i$ for all i .
fparmacs_operator2	$F1_i = F1_i + F2_i A + B$ for all i .
fparmacs_operator2v	$F1_i = F1_i + F2_i A_i + B_i$ for all i .

Chapter 7

PROCESS ENGINEERING LANGUAGE - BASIC PRINCIPLES

7.1 Basic Concepts of PEL

PEL (Process Engineering Language) is a problem oriented-language by which the user instructs and communicates with any template based system (TBS) created using GPES.

The basic elements of the language are commands (statements). Each command is a request for an action to be taken by the system. The system accepts a command, interprets it, and accomplishes what the user intended by that command. Then the system is ready to accept another command. The system ignores and prints error messages for illegal commands. Each command begins with a keyword called a command verb and may be followed by another keyword called the command object and is followed by the command body (if any) and will be terminated by a semicolon. The command verb indicates the function of the command and the command object indicates the object upon which the action should be taken. Table 7.1 shows the command verbs, and the corresponding command objects. This table is the output of PEL command: Help commands;. There is no reserved word in PEL. Consequently there is no need for the user to remember a long list of reserved names which he must learn to avoid, even if he does not know what they mean.

7.1.1 Command Objects

A process may be viewed as a collection of equipment units, and streams containing masses of individual chemical species, momenta, and energy which flow through the process. To model and design a process it is necessary to represent these elements of the process. Some of these

TABLE 7.1 PEL COMMANDS

**COMMAND {help commands}

		Pel commands						
command	verbs abrevia- tion	unit units	component components	function functions	command objects			Process Processes
					stream streams	flow f	variable variables	
assume	a	x	x	x	x	x	x	
buss								
calculate	calc	x	x	x	x			
clear	clr							
close	cl		x					x
connect	cnct	x						
continue	ct							
copy			x					
create	crt	x	x	x	x	x		
delete	del	x	x	x	x	x		
deletef	delf		x					x
disconnect	disc	x						
end								
escape	esc							
help	h							
include	inc		x					
leave	lv							
let		x	x	x	x	x		
leta		x	x	x	x	x		
list	l	x	x	x	x		x	
listf	lf		x					x
listt	lt	x	x	x	x			
load	ld		x					x
loop	lp							
news								
open			x					x
Print	P	x	x	x	x	x	x	x
Printf	Pf		x					
Printt	Pt							
Profile	Prof							
read	rd	x	x	x	x	x	x	
reada	rda	x	x	x	x	x	x	
repeat	r							
save	sv		x					x
specify	sp	x	x	x	x	x	x	
stop								
terminate	term		x					x
unspecify	unsp	x	x	x	x	x	x	
use								

note: for user defined function only "create", "delete" ,and "list" commands apply.

elements are basic, such as units, streams, components, and some are auxiliary such as variables and functions to enhance the users calculating ability. Commands are used to create and manipulate these elements.

Command objects represent these elements and are as follows:

unit

A process unit may be thought of as an entity with the following attributes:

NAME: Which identifies the process unit.

TYPE: Which determines the function of the process unit (e.g., MHD generator, heat exchanger, etc.)

INLET STREAMS: Those streams that are entering the process unit.

OUTLET STREAMS: Those streams that are leaving the process unit.

PARAMETERS: Operating variables and other characteristics of the unit.

stream

In general a stream may be thought of as an entity with the following attributes:

NAME: Which identifies the stream.

TYPE: The type of the stream (e.g., liquid vapor stream, information stream, etc.)

SOURCE: The unit from which the stream originates (a stream need not originate from a unit. It may also enter from the environment external to the process).

DESTINATION: The unit to which the stream is entering (a stream need not enter into a unit. It may also enter to the environment external to the process).

ONE OR MORE PHASES:

Each phase may be characterized by the following attributes:

FLOW: Flow information of the components present in the phase.

The command object "flow" is used to represent such information.

PHASE (STREAM) PARAMETERS:

Thermodynamic state variables and those physical/chemical/transport properties used to define the stream.

Usually phase 0 represents the total stream.

component

A chemical component may be considered as an entity with the following attributes:

NAME: The name of the component.

TYPE: Type of the component (indicates number of parameters, etc.)

PARAMETERS: Physical properties of the component.

flow

One or more components may flow in a stream. Information regarding the flow may be characterized by the following attributes:

COMPONENT - NAMES: The name of each component present in a stream .

For each phase of the stream:

FLOW PARAMETERS: There are some parameters of a phase which are specific to a component in a phase. We refer to these component related parameters as "Flow Parameters" (e.g., Flow Rate, Mole Fraction,

Diffusion Coefficients, etc). For each component present in the stream, these parameters represent the flow of the component in the phase.

function

There are three types of functions available to the user. They are:

pre-defined functions

user-defined functions

built-in functions

A function can be referenced in an expression as follows: function-identifier (expression, ...) to represent a value which is the result of the function evaluation. A short description of each type of functions follows. A detailed discussion of functions is given in Appendix D .

Pre-Defined Functions

These are the functions defined and established in a template based system by its system administrator by providing the appropriate templates and required subprograms.

A pre-defined function may be considered as an entity with the following attributes:

NAME: The name of the function.

TYPE: The type of the function (indicates number of parameters, number of arguments, routine used to evaluate it, etc.).

PARAMETERS: Coefficients of the function.

User-Defined Functions

These are the functions that the user may provide at will to improve his problem-solving capability.

Such a function may be considered as an entity with the following attributes:

NAME: The name of the function.
 ARGUMENTS: The list of dummy arguments.
 EXPRESSION: The arithmetic expression representing the function.

Built-In Functions

These are functions built into the system and available to the user.

variable

A variable is an entity which represents a value. The variables may be classified to the following groups:

- 1) Simple or user-supplied variables (e.g., X,Y,Z).
- 2) Qualified variables. A qualified variable is a variable referring to a parameter of a user-created unit, stream, component, flow, or a pre-defined function. A unit qualified variable refers to a parameter of a unit, for example u.A.N. refers to parameter "N" of unit "A". A component qualified variable refers to a parameter of a component, for example c.CO.MW refers to parameter "MW" of component "CO". A function qualified variable refers to a parameter (coefficient) of a pre-defined function, for example fn.H.A0 refers to parameter "A0" of function "H". A stream qualified variable refers to a parameter of a specific phase of a stream, for example, s.FEED.p0.T refers to parameter "T" of phase zero of stream "FEED". A flow qualified variable refers to a flow parameter of a component in a specific phase of a stream, for example f.FEED.p0.CO.X refers to flow parameter "X" of component "CO" in phase zero of stream "FEED".
- 3) Built-In variables (Built-in constants). A built-in variable when appearing in an arithmetic expression will represent a pre-set value and it will be replaced by its value when used. In this

sense, it is more appropriate to call it a built-in constant. The built-in constants currently implemented are listed in Table 7.2. Such a variable is only interpreted as a built-in constant if there is no other user-supplied variable with the same name.

process

A process represents the collection of above elements which have been created by a user. When a user enters the system a working area will be established to contain the created process elements. The content of this working area is called the active or current process or the process in the working area. The user may save this process and later retrieve it for further investigation and analysis.

7.1.2 Process Files

A process file is a file which may contain any number of processes. The user may create any number of process files. The user may save his active process (process in the working area) in a process file or retrieve a process from the file and store it in the working area. Users may share their process files.

7.1.3 Component Files

A component file is a file which contains the physical properties of components, or constants required by property estimation routines to calculate physical properties. The components in the file may be of one or more types. A user may create any number of component files, and users may share their private component files. The TBS administrator may also create such files and provide the users with access to them. Such files that are accessible to all users are called public component files.

7.1.4 Property Estimation Methods

A TBS may offer more than one method for estimation of each physical

Table 7.2

Built-In Constants

<u>Name</u>	<u>Value</u>	<u>Representing</u>
%e	2.718281828	Base of Natural Logarithm
%f	9.64870X10 ⁴	Faraday Constant in coul/mole
%n	6.02252X10 ²³	Avagadro's number in number/mole
%pi	3.141592657	Pi
%r	1.98719	Gas constant in Kcal/ ^o K-Kgmole

property. The user may choose the method to be used for one or more of these properties. Default options will be used for those properties which have not been specified by the user.

7.1.5 Units of Measurement

A TBS may allow the user to provide his data in one or more units of measurement. The unit of measurement must be enclosed in a pair of apostrophes and should immediately succeed the item of data to which it refers. If no unit of measurement is provided for an item of data the proper standard unit of measurement for that item of data will be assumed.

7.1.6 Profile Parameters

These parameters either control the format of output, or they specify the input and output files to be used by the system. The TBS provides the default values for these parameters. The user may change these parameters at any time.

Profile parameters are as follows:

sdigit - Significant number of digits to be printed for numerical data.

ddigit - Decimal number of digits to be printed for numerical data.

dflag - Debugging flag indicates whether additional information regarding command processing is to be printed.

output - Indicates the output file.

input - Indicates the input file.

7.1.7 Arithmetic Expressions

Arithmetic expressions may be used freely in a PEL command whenever a numerical data is expected. An arithmetic expression is a representation of a value. A number, a simple variable, or a qualified variable is an expression. Combinations of numbers, simple variables, qualified variables and/or functions, along with operators (e.g. +, -, *, **, etc) are also

expressions. A full description of arithmetic expressions in PEL is given in Appendix D.

7.1.8 Command Elements

There are very few restrictions in the format of PEL commands. Consequently instructions can be written without consideration of special coding forms. As long as each command is started on a new line and terminated by a semicolon (;), the format is completely free. A command is constructed from symbols. A symbol is the string of one or more characters (up to 16 characters). There are three types of symbols: Literals, Terminal Symbols (operators), and Identifiers.

Examples of Literals are 123, 123.475, and 12.3e+1. Examples of Terminal Symbols are +, -, *, >, & , and = . Identifiers can be divided into three groups: Language Keywords, Established Identifiers, User-supplied Identifiers. A Language Keyword is an identifier that, when used in proper context, has a specific meaning to the system. Established Identifiers are the keywords of the TBS. These identifiers have been set in the TBS by its responsible system administrator. User-supplied identifiers are the identifiers provided by the user to identify the objects he creates. Note that there is no reserved word in PEL. Language Keywords and Established Identifiers will be interpreted as such only when used in proper context.

For example the following command:

```
create unit (create,heatx) type = heatx; will create two units of type
"heatx". One unit is named "create" and another one is named "heatx".
Notice that only the first appearance of "create" is interpreted as a
Language Keyword. Similarly the second appearance of "heatx" is interpreted
as an Established Identifier.
```

A detailed description of command elements is given in Appendix D.

7.2 Classification of Commands

Each command is a request for an action to be taken by the system. Commands enable a user to create process elements, specify parameters and variables, calculate parts of or whole flowsheet, print results, etc. When a user enters the system a working area will be established to contain the created process elements. The content of the working area is called the active process or the current process. The user can save this active process in a process file and later retrieve it for further investigation.

In this section commands are classified according to their function. A command may appear in more than one group. Each command is followed by a short description. A more detailed description and command syntax is given in Appendix D.

7.2.1 Configuration Commands

These commands may be used to specify the process configuration:

- | | |
|------------|--|
| create | To create units, components, functions (pre-defined and user-defined), streams, or to specify what components are flowing in a stream (flow). |
| delete | To delete (remove) units, components, functions (pre-defined and user-defined), streams, or to deny the flow of some components in a stream (flow). |
| connect | To connect a unit to streams. |
| disconnect | To disconnect a unit from streams. |
| let, leta | To duplicate or copy a unit, component, function (pre-defined function), or stream. |
| load | To copy the components from public files or user's private files into the user's working area, or to copy an entire process from a user's process files into his working area. |

7.2.2 Value Assignment Commands

These commands may be used to assign values to parameters, variables, etc.

- specify** To specify the value of unit parameters, component parameters, function parameters (pre-defined functions), streams parameters (phase or flow parameters), or variables.
- unspecify** To unspecify the value of parameters or variables.
- assume** Identical to the "specify" command with the exception that the value assigned is assumed rather than specified.
- let,leta** To duplicate or copy unit, component, function (pre-defined function), or stream.
- calculate** To calculate units, components, functions (pre-defined functions) or streams. The GPES executive will call upon the associated subprograms from the attached template based system to perform the calculations.
- read,reada** To accept data from the terminal and to assign it to variables, and parameters.
- repeat** It is the first command of a sequence of commands, and specifies repetitive execution of the commands within the sequence. It also assigns a value to its control variable.
- copy** To copy the component parameters from public component files to a user's private component files.
- include** To accept data from the terminal and assign it to parameters of specified components in the user's private component files.
- use** To instruct the system to use specified methods of physical property estimations.
- profile** To allow the user to provide his/her desired profile parameters.

7.2.3 Output Commands

These commands are used to display information:

- `print` To print information about units, components, functions (pre-defined), streams, flow, variables, or the entire process in the working area.
- `list` To list names of units, components, functions (pre-defined and user defined), streams, or variables existing in the working area.
- `printf` To print parameters of specified components from the component public files or component private files.
- `listf` To list the names of components on component public files or component private files.
- `printt` To print information about the template of a unit type, stream type, component type, function type, (pre-defined type), or other information regarding the template data base.
- `listt` To list the types of units, streams, components, functions, (pre-defined) available in the template data base.
- `use` To print the physical property estimation methods which are in effect.
- `profile` To print the profile parameters in effect.
- `help` To instruct the system to provide a description of a specific command or object, or to list the PEL commands.
- `news` To obtain current information about GPES and the TBS.
- `bugs` To print the reported errors regarding GPES and the TBS.

7.2.4 Input Commands

These commands may be used to input data.

- `read,reada` To accept data from the terminal and to assign it to variables and parameters.

include To read component data from a terminal and store the information in a component private file.

7.2.5 Clearing and Switching Commands

These commands may be used to clear the working area and/or to switch to another template based system or to the Multics level.

clear To clear the working area.

load It may be used to clear the working area and to load another process from a private process file into the working area.

escape Allows the user to execute any number of Multics commands.

leave To leave the current attached template based system and attach another one.

end To terminate the program.

7.2.6 Commands for Component Files

A user may have any number of component files which may be shared by other users. Each TBS may have any number of public component files accessible to the users of that TBS. The commands listed in this group are used for data management and accessing these files.

open To introduce the desired private or public file to be used in subsequent commands or to create a new private file. Only one private file and one public file can be opened at any one time.

close To undo the effect of the open command.

terminate To terminate (destroy) the specified private files.

save To copy the components from the working area into the opened private file.

load To copy the components from the opened private or public files into the working area.

deletef To delete the specified components from the opened private file.

- `listf` To list the components of a specified type existing on the opened public or private file.
- `printf` To print the parameters of specified components on the opened public or private file.
- `copy` To copy components from the opened public file into the opened private file.
- `include` To include components whose parameters are provided by the user from the terminal into the opened private file.

7.2.7 Commands for Process Files

A user may have any number of process files which may be shared by other users. A process file may contain any number of processes. Commands listed in this group are used for data management and accessing of these files.

- `open` To introduce the desired process file to be used in subsequent commands, or to create a new process file.
- `close` To undo the effect of open command.
- `terminate` To terminate (destroy) the specified process files.
- `save` To copy the process from the working area into the opened process file.
- `load` To copy the specified process from the opened process file into the working area.
- `deletef` To delete specified processes from the opened process file.
- `listf` To list the processes existing in the opened process file.

7.2.8 Continue Command

It is a null command which does not perform any action.

7.2.9 Iterative Commands

These commands may be used to perform repetitive execution of a sequence of commands.

repeat Heads a sequence of commands and specifies repetitive execution of the commands within the sequence.

loop Designates the end of the sequence of commands headed by a repeat command.

stop Aborts repetitive execution of a group of commands headed by a repeat command.

The following commands are not allowed to appear inside a repeat-loop group:

delete
let
leta
unspecify

The following commands will clear the working area and will terminate the execution of the repeat-loop commands:

clear
end
leave
load process

The following commands when appearing inside a repeat-loop sequence will be executed only once, independent of the condition of the repeat-loop:

bugs	listf
close	listt
connect	load component
continue	news
copy	open

create	printf
deletef	printt
disconnect	profile
escape	save
help	terminal
include	use
list	

Only the following commands when appearing inside a repeat-loop sequence will be executed repeatedly depending on the condition of the loop:

assume
 calculate
 loop
 print
 read
 reada
 repeat
 specify

7.3 Using a TBS

To use a TBS, the user should register with the TBS Administrator. When entering the GPES executive the user is prompted for the name of the TBS. When using PEL commands the user may exit from one TBS and enter another one, or print any information regarding the entered TBS.

7.4 Using the System

In using the system the following steps are taken:

Step 1: Entering the system:

- a. When the user is in the Multics ready state he should type in one of the following:

pel

pel brief

pel bf

The function of the argument brief or bf is to inhibit the printout of messages about the system and the attached TBS.

- b. If the brief argument is not provided, some information about the system will be printed.
- c. If the GPES system administrator has locked the system, a system message will be printed and the execution of the program will be terminated.

Step 2: Attaching a TBS:

- a. The user will be prompted to provide the name of the TBS he wishes to use. To be able to use a TBS he should have been authorized by that TBS system Administrator.
- b. If the user is not an authorized user of the specified TBS or has specified an unknown TBS, a system message will be printed and he will be given the option of trying another TBS or leaving the system.
- c. If the brief argument was not provided in Step 1, some information about the TBS will be printed.
- d. If the TBS system administrator has locked the TBS, a message will be printed and the user will be given the option of trying another TBS or leaving the system.
- e. If the TBS is inconsistent, a message will be printed and the user will be given the choice of continuing to use the TBS or not using it. If he chooses the latter he will be given the choice of trying another TBS or leaving the system.

- f. If any component file is opened it will be closed.
- g. Profile parameters will be initialized to the default parameters of the TBS.

Step 3: Creating a New Process:

- a. A new process will be created.
- b. All property estimation options will be initialized to default parameters of the attached TBS.
- c. The user will be prompted to provide the maximum number of components.

Step 4: Command Level:

The user is now at command level. He may issue any of 39 PEL commands. The system executes that command and then it is ready to accept another command. The user will remain at the command level except for the following commands:

leave: Clears the working area and transfers control to step 2.

escape: Transfers control to the Multics command level, where the user may execute any number of Multics commands. To return to PEL command level, the user should enter a blank line (press the return key on the terminal).

clear: Clears the working area and transfers control to step 3.

end: Terminates the session.

7.5 PEL Messages

PEL messages are those messages that a user may receive when using the GPES executive. Appendix D lists these messages and gives an example of each occurrence of these messages. In this section, a general description of PEL messages is given. PEL messages can be grouped into the following five classes:

- 1) Information Messages
- 2) Requesting Messages
- 3) Warning Messages
- 4) Error Messages
- 5) System Messages

Warning messages and system messages each are divided into two types: informatory and severe. Error messages are divided into three types: informatory, severe, and calculate. Table 7.3 presents the format of PEL messages.

7.5.1 Information Messages

These messages are not the result of a wrong action on the part of the user. They only provide information about the system or the status of the operation. They are in the following form:

*information message.

7.5.2 Requesting Messages

These messages prompt the user to provide input or to answer a question. They are in the following form:

**requesting message:

The user should then enter the requested data or his response to the question.

7.5.3 Warning Messages

These messages call attention to a possible area of negligence on the part of the user. They do not imply syntactical or other types of errors in the command. Usually the execution of the command will continue after detection of the condition evoking these warnings. Warning messages are of two types: informatory and severe.

Table 7.3Format of PEL Messages

*Information Message.

**Requesting Message:

WARNING i nn informatory warning message.

WARNING s nn severe warning message.

ERROR i nn informatory error message.

ERROR s nn severe error message.

ERROR c nn calculate error message.

SYSTEM i nn informatory system message.

SYSTEM s nn severe system message.

Where:

i indicates that the message is of informatory type,

s indicates that the message is of severe type,

c indicates that the message is of calculate type,

nn is the appropriate message number.

7.5.3.1 Informatory Warning Messages

These messages may be produced during the execution of a command. The execution of the command will continue after the issuance of such warnings. They usually do not require responsive actions. The presence of these messages indicates that the command is not fully executed. The execution of the parts skipped is not permitted, because it violates some restrictions or the desired actions have already been accomplished.

7.5.3.2 Severe Warning Messages

These messages may be provided during the evaluation of an expression. The conditions which cause these messages cannot usually be detected before execution of the command. For this reason, a message will be printed and the result of the expression will be assumed to be zero, and execution of the command will be continued. Although the system does not require responsive action, the user may want to undertake some corrective actions once the execution of the command(s) has been completed. Table 7.4 presents the conditions in an expression producing severe warning messages.

7.5.4 Error Messages

These messages are the result of syntactical errors or other types of errors made by the user. Error messages are of three types: Informatory, Severe, and Calculate.

7.5.4.1 Informatory Error Messages

These are errors detected in the user's input. The system prints the message and asks the user to reenter his input. Because these errors could be corrected by the user the execution of the command will be continued. They are the least severe type of errors. Table 7.5 indicates the conditions under which these error messages may be produced.

Table 7.4

conditions in an expression producing severe warning messages

severe warning message number	condition
1	$ x > 1$ in <code>acos(x)</code> or <code>asin(x)</code>
2	$ x \geq 1$ in <code>atanh(x)</code>
3	$x \leq 0$ in <code>log(x)</code> , <code>log2(x)</code> , or <code>log10(x)</code>
4	$x = 0$ in <code>mod(y,x)</code>
5	$x < 0$ in <code>sqrt(x)</code>
6	$x = 0$ & $y = 0$ in <code>atan(x,y)</code> or <code>atand(x,y)</code>
7	zerodivide
8	overflow
9	underflow
10	$x \leq 0$ in <code>0.00 ** x</code>
11	one or more parameters of a pre-defined function appearing in the expression are unspecified.

Table 7.5

States Where Informatory Error Messages may be Produced

Informatory
Error
Message

<u>Number</u>	<u>May Occur in:</u>
1	inputing a line of characters or a command line
2-15	inputing a command line.
16-17	inputing an integer number
18	inputing a (real) number
19	inputing yes or no
20-22	inputing in response to a read, or reada command

7.5.4.2 Severe Error Messages

Severe errors are the result of syntactical errors or other kinds of errors that cannot be corrected. Upon the first detection of such an error, the error-checking procedure will be discontinued and the execution of the command will be ignored.

7.5.4.3 Calculate Error Messages

These errors are detected during the execution of a calculate command. The system responds differently to this type of error. Errors of this type are only detectable during the execution of the command while errors of type "severe" are detected before the execution of the command. Once an error of type "calculate" has been detected in a command the execution of that command will be halted. If the command resides in a closed loop the execution of all commands in the loop(s) will be stopped. If the command is the last command typed by the user (e.g., not in a loop or in an open loop) only the execution of that command will be stopped and preceding commands if any will be unaffected.

7.5.5 System Messages

System messages are issued in response to conditions which are not necessarily attributable to the user's actions. They may be the result of negligence on the part of the GPES Administrator, TBS Administrator, or the user. These messages may be produced upon entering the system or attaching a TBS or executing a command. They indicate conditions such as incompatibility, improper access, presence or absence of some segments (files). The user may be responsible in the following cases:

- a) Attaching a TBS which is either:

Unknown, inconsistent, or inaccessible to the user.

- b) Opening a process file, or opening a component file, or loading a

process which may not be compatible with the current version of the system or the attached TBS.

- c) Not having the required access for using a component or a process file, or providing invalid path names for that purpose.
- d) Intervening in the system file management by deleting or modifying segments the system has created for him, or creating such segments outside of the system. The names of these segments are always suffixed by ".GPES".
- e) Attempt to use a proprietary or an unknown routine.

System messages are of two types: `informatory` and `severe`.

7.5.5.1 Informatory System Messages

These messages may be produced upon entering the system, attaching a TBS, or executing a command. The operation of the system or execution of the command usually will continue after the issuance of such messages. They usually do not require responsive actions. When produced as a result of the execution of a command. They indicate that the command is not fully executed. The execution of the parts skipped is not permitted because it violates some restrictions or provides invalid information or refers to unknown segments.

7.5.5.2 Severe System Messages

These messages may be produced upon entering the system, attaching a TBS, or executing a command. If detected during entrance to the system, admission will be denied. If detected during attachment of a TBS, attachment of that TBS will be stopped. If detected during the execution of a command, that command will be ignored.

CHAPTER 8

GPES ADMINISTRATION AND PROTECTION

GPES may contain any number of TBS's, each having any number of users. To assist the GPES administrator in administrating and protecting the system and each TBS from unauthorized users, a mechanism has been provided as a part of GPES, to protect files and insure security. GPES has three files for this purpose: Text file, TBS table, and users table. The text file also serves other purposes as will be described next.

8.1 The GPES Text File

The GPES text file is created and updated by any available editor. It contains the following information:

- a) The GPES Administrator's name.
- b) Any reference about the GPES.
- c) Any message to the users.

The above information (a,b,c) will be printed when the user enters the system.

- d) The system lock which indicates whether the system can or cannot be accessed by the users.
- e) Any news about the GPES. This information along with similar information about the attached TBS will be printed in response to the user's "news" command.
- f) Any errors found or reported about GPES. This information along with similar information about the attached TBS will be printed in response to the user's "bugs" command.
- g) Information describing the PEL commands which will be printed in response to the user's "help" commands. It contains the following:

- 1) Syntax notation conventions.
- 2) A description of each of the seven objects: unit, component, function, stream, flow, variable, and process.
- 3) A description of each PEL command followed by all its subgroups.

The format of the file is shown in Figure 8.1. Each item of information should start and terminate by a %. The heading for each item should be exactly as shown in Figure 8.1. Notation ---text--- indicates that any number of characters or lines can be provided. Although the sequence of providing the items of information is not important, the sequence shown in Figure 8.1 is recommended. The format of "lock item" should be one of the following:

```
%lock open%
```

```
%lock close%
```

If this item is not provided or it is as the first format, the system can be accessed. The second format indicates that the system cannot be accessed.

8.2 The TBS Table

This table contains information regarding every TBS, as shown in Figure 8.2. Each entry of the table may be empty or contain information regarding a TBS. Each entry contains the following information:

- a) The date on which the TBS was added to the table.
- b) The TBS's name.
- c) The pathname of the directory containing the template data base.

This information enables the GPES executive to locate the template data base once the user provides the TBS' name.

- d) The TBS administrator's name, address and telephone number.

Figure 8.1

Format of The GPES Text File

```

%*GPES message ----Text----%
%*GPES reference ----Text----%
%*GPES system administrator----Text----%
%lock open% or %lock close%
%GPES news----Text----%
%GPES bugs----Text----%
%syntax notation conventions----Text----%
%unit----Text----%
%component----Text----%
%function----Text----%
%stream----Text----%
%flow----Text----%
%variable----Text----%
%process----Text----%
%assume----Text----%
%assume unit----Text----%
%assume component----Text----%
%assume function----Text----%
%assume stream----Text----%
%assume flow---Text---%
%assume variable---Text---%
%bugs---Text---%
%calculate---Text---%
.
.
.
%use---Text---%

```


TBS TABLE

	SYSTEM'S NAME (i.e., GPES)
	NUMBER OF ENTRIES = n

	DATE
	TBS NAME
	THE PATHNAME OF THE DIRECTORY CONTAINING THE TEMPLATE DATA BASE
	TBS ADMINISTRATOR'S NAME
	TBS ADMINISTRATOR'S ADDRESS
1	TBS ADMINISTRATOR'S TELEPHONE NUMBER
	CURRENT USAGE
	TOTAL USAGE
	PENDING FLAG
	INUSE FLAG

.	
.	
.	
n	

FIGURE 8.2 THE TBS TABLE

- e) The TBS current and total usage. A mechanism for recording this information has been proposed, but has not been implemented. It will be described in Chapter 11.
- g) The pending flag which indicates whether the TBS is not currently operational, or could be used.
- h) The inuse flag which indicates whether the entry is empty or not. It is also a safety flag which is "on" when the entry contains complete information about a TBS.

An interactive program called "update_tbs_tbl" has been developed to create and update this file. The GPES Administrator communicates with this program by a simple set of commands. The command syntax is as follows:

```
command [tbs-name]
```

For instance, if the GPES Administrator wants to insert a TBS called "XYZ" the command for it would be: insert XYZ, or in the abbreviated form: i XYZ. Having received the insert command, the program prompts the user for all the required information. The complete list of commands is given in Table 8.1. Using an "*" as the tbs_name indicates that the command must be executed for each TBS in the Table. Before the program accepts any command it would create a new table if one does not already exist.

The function of each command is as follows:

insert: To insert a new TBS into the table. The program prompts the GPES Administrator for all the required information. If the TBS Administrator is new (if he is not the administrator of an existing TBS), the program will also provide the TBS

Table 8.1

Commands Used by the GPES Administrator to Update the TBS Table

<u>Command</u>	<u>tbs_name</u>
1) insert or i	tbs_name
2) delete or d	tbs_name or *
3) print or p	tbs_name or *
4) pend or pd	tbs_name or *
5) restore or r	tbs_name or *
6) end	

Administrator access to programs he may need in performing his duties (i.e., update_tdb program and other utility programs).

delete: To remove a TBS from the table. If the TBS Administrator is not the Administrator of any other existing TBS, the program also denies the access rights given earlier.

print: To print information regarding a TBS.

pend: To change the status of a TBS to pending, so that the TBS cannot be accessed by its users.

restore: To undo the effect of the pend command.

end: To terminate the program.

The GPES administrator will add an entry to the table upon receiving the TBS registration form shown in Figure 8.3.

8.3 The Users Table

This table contains information regarding every user of each TBS. As shown in Figure 8.4, the table consists of a number of data structures chained together. Each of these data structures represents a user of a TBS. A user may have access to one or more TBS. Therefore associated with each user there are a number of such data structures, each corresponding to a TBS for which the user has authorization for using the TBS. Each data structure contains the following:

- a) A pointer to the next data structure in the list.
- b) A pointer to the previous data structure in the list.
- c) Date on which entry was made.
- d) User's name.
- e) TBS name.
- f) User's address, and telephone number.
- g) User's current and total usage of the TBS. A mechanism for

FIGURE 8.3
TBS REGISTRATION FORM

TO: GPES ADMINISTRATOR	FROM:								
DATE	<table border="1" style="width: 100%; height: 20px;"> <tr> <td style="text-align: center;">TBS NAME</td> </tr> </table>	TBS NAME							
TBS NAME									
REQUEST FOR:	<table style="width: 100%;"> <tr> <td style="text-align: center;">ADDING</td> <td style="text-align: center;">DELETING</td> <td style="text-align: center;">PENDING</td> <td style="text-align: center;">RESTORING</td> </tr> <tr> <td style="text-align: center;"><input type="checkbox"/></td> <td style="text-align: center;"><input type="checkbox"/></td> <td style="text-align: center;"><input type="checkbox"/></td> <td style="text-align: center;"><input type="checkbox"/></td> </tr> </table>	ADDING	DELETING	PENDING	RESTORING	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
ADDING	DELETING	PENDING	RESTORING						
<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>						
CHECK ONE:									
<p><u>ADDING</u></p> <table border="1" style="width: 100%; height: 20px;"> <tr> <td style="text-align: center;">TBS ADMINISTRATOR'S PERSON_ID</td> </tr> </table> <table border="1" style="width: 100%; height: 20px;"> <tr> <td style="text-align: center;">TBS ADMINISTRATOR'S ADDRESS</td> </tr> </table> <table border="1" style="width: 100%; height: 20px;"> <tr> <td style="text-align: center;">TBS ADMINISTRATOR'S TELEPHONE NUMBER</td> </tr> </table> <table border="1" style="width: 100%; height: 20px;"> <tr> <td style="text-align: center;">PATHNAME OF THE DIRECTORY CONTAINING THE TEMPLATE DATA BASE</td> </tr> </table> <table border="1" style="width: 100%; height: 20px;"> <tr> <td> </td> </tr> </table> <table border="1" style="width: 100%; height: 20px;"> <tr> <td> </td> </tr> </table> <table border="1" style="width: 100%; height: 20px;"> <tr> <td> </td> </tr> </table> <table border="1" style="width: 100%; height: 20px;"> <tr> <td> </td> </tr> </table>		TBS ADMINISTRATOR'S PERSON_ID	TBS ADMINISTRATOR'S ADDRESS	TBS ADMINISTRATOR'S TELEPHONE NUMBER	PATHNAME OF THE DIRECTORY CONTAINING THE TEMPLATE DATA BASE				
TBS ADMINISTRATOR'S PERSON_ID									
TBS ADMINISTRATOR'S ADDRESS									
TBS ADMINISTRATOR'S TELEPHONE NUMBER									
PATHNAME OF THE DIRECTORY CONTAINING THE TEMPLATE DATA BASE									
A BRIEF DESCRIPTION OF THE TBS:	APPROVED BY:								

USERS TABLE

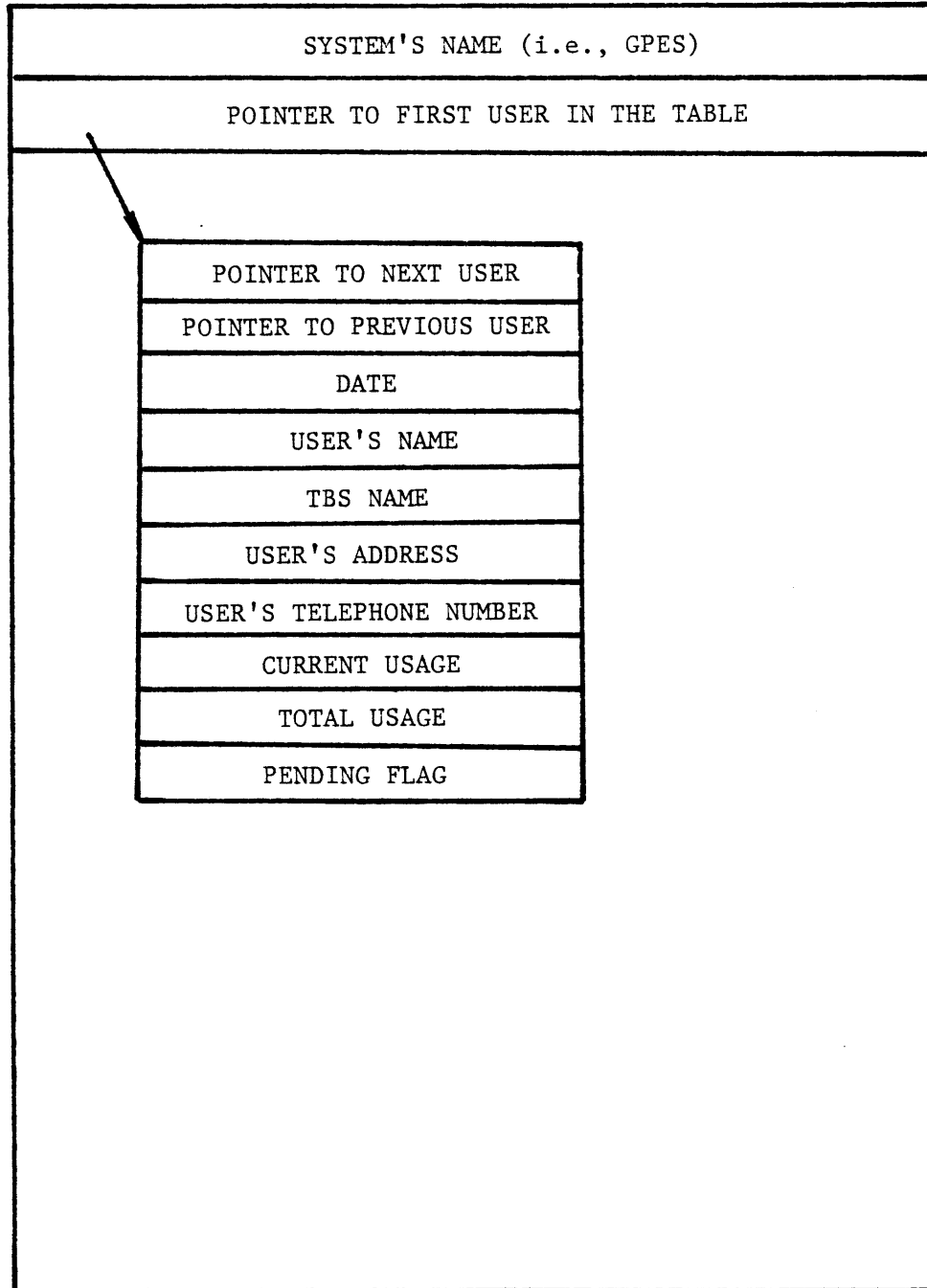


FIGURE 8.4 THE USERS TABLE

recording this information is proposed, but has not been implemented. It will be discussed in Chapter 11.

- h) Pending flag which indicates the status of the user regarding the usage of the TBS. When it is "on" the user can not use the TBS.

The data structures in the list are alphabetically ordered by users' names. The data structures for the same user are ordered by TBS's names.

An interactive program called "update_users_tbl" has been developed to create and update this table. In fact, both update_tbs_tbl, update_users_tbl are entries of a single program, called the Administrative Program. This program has also two other entries, used by the GPES executive to obtain information about a TBS or to verify whether the user is authorized to use the requested TBS.

The GPES Administrator communicates with the update_users_tbl program with a simple set of commands. The command syntax is as follows:

```
command [user_name] [tbs_name]
```

For example, if the GPES Administrator wants to authorize user "Smith" to use the TBS "XYZ" he would give the command:

```
insert Smith XYZ
```

Having received the insert command, the program prompts the user for all the required information. The complete list of commands is given in Table 8.2. Using an "*" as the tbs_name or the user_name indicates that the command must be executed for all TBS's or all users. Before the program accepts any command it creates a new "users table" if one does not already exist.

The function of each command is as follows:

insert: To make an entry in the table (to authorize a user for a TBS).

The program prompts the GPES Administrator for all the required

Table 8.2

Commands used by the GPES Administrator to Update the Users Table

<u>Command</u>	<u>User_name</u>	<u>TBS_name</u>
1) insert or i	user_name	tbs_name
2) delete or d	user_name or *	tbs_name or *
3) print or p	user_name or *	tbs_name or *
4) pend or pd	user_name or *	tbs_name or *
5) restore or r	user_name or *	tbs_name or *
6) end		

information. The program also gives the new user necessary access to the GPES executive.

delete: To delete an entry in the table (to delete a user from the list of authorized users of a TBS). If the user is not using any other TBS, the program would also deny the access rights given earlier, when the user was first inserted to the table.

print: To print information about a user of a TBS.

pend: To prevent the user from using the TBS, temporarily.

restore: To undo the effect of the above command.

end: To terminate the execution of the program.

The GPES Administrator will add an entry to the table upon receiving the user registration form shown in Figure 8.5.

8.4 Various Copies of GPES Files

Two copies of each GPES file are required and a third one is recommended. The two required copies are:

- 1) The original or primary copy accessible to the GPES Administrator only.

Table 8.3 lists the names of segments containing the primary copy of GPES files.

- 2) The system copy accessible to the GPES Executive program. The segment names for this copy are the same as in Table 8.3, except that they are suffixed by ".syscopy". The system copy can be used while the primary copy is being updated. The users have access to this copy only.

A backup copy is also recommended, to enable recovery from unintentional erasures and other accidents. The segment names for the backup copy are the same as shown in Table 8.3, except that they are suffixed by ".backup". The flow of information between the different copies of the GPES files, the GPES Executive, and other utility programs is

FIGURE 8.5
GPES USER REGISTRATION FORM

TO: GPES ADMINISTRATOR		FROM: TBS ADMINISTRATOR		
DATE	<input type="text"/> TBS NAME			
REQUEST FOR:	ADDING	DELETING	PENDING	RESTORING
CHECK ONE:	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
ADD				
<input type="text"/> USER'S PERSON_ID				
<input type="text"/> USER'S ADDRESS				
<input type="text"/>				
<input type="text"/> USER'S TELEPHONE NUMBER				
DELETE, PEND, OR RESTORE				
<input type="text"/> USER'S PERSON_ID				
OR CHECK THIS BOX IF THE REQUEST IS FOR ALL USERS <input type="checkbox"/>				
APPROVED BY:				

Table 8.3

GPES Files (primary copy)

<u>Segment Name</u>	<u>Contents</u>	<u>Created and Updated by</u>
stext	text file	any available editor
tbs_tbl	tbs table	update_tbs_tbl program
users_tbl	users table	update_users_tbl program

shown in Figure 8.6. The utility program "copy_seg" copies one segment into another, while preserving the access control list of the target segment. Access control list of a segment contains the name of authorized users of the segment. Utility programs are described in Chapter 4.

8.5 GPES Organization

There are four levels of activity associated with GPES. The hierarchy is represented in Figure 8.7. Each level is the responsibility of a different set of personnel:

- 1) The GPES Administrator who is responsible for:
 - a) Maintenance of the GPES.
 - b) Permitting a TBS Administrator to use the system.
 - c) Permitting the users introduced by a TBS Administrator to use the system.
- 2) TBS Administrators are responsible for:
 - a) Implementation and maintenance of template based systems.
 - b) Coordination of TBS programmers.
 - c) Permitting a user to use a TBS.
- 3) TBS Programmers are responsible for development of TBS programs.
- 4) Users who are the ultimate users of the system, the designers of chemical processes.

For each of these groups some tools have been developed to assist them in performing their duties. The tools are a set of programs and languages to be used to communicate with those programs.

Table 8.4 lists the various programs that an individual in each group requires to perform his duties.

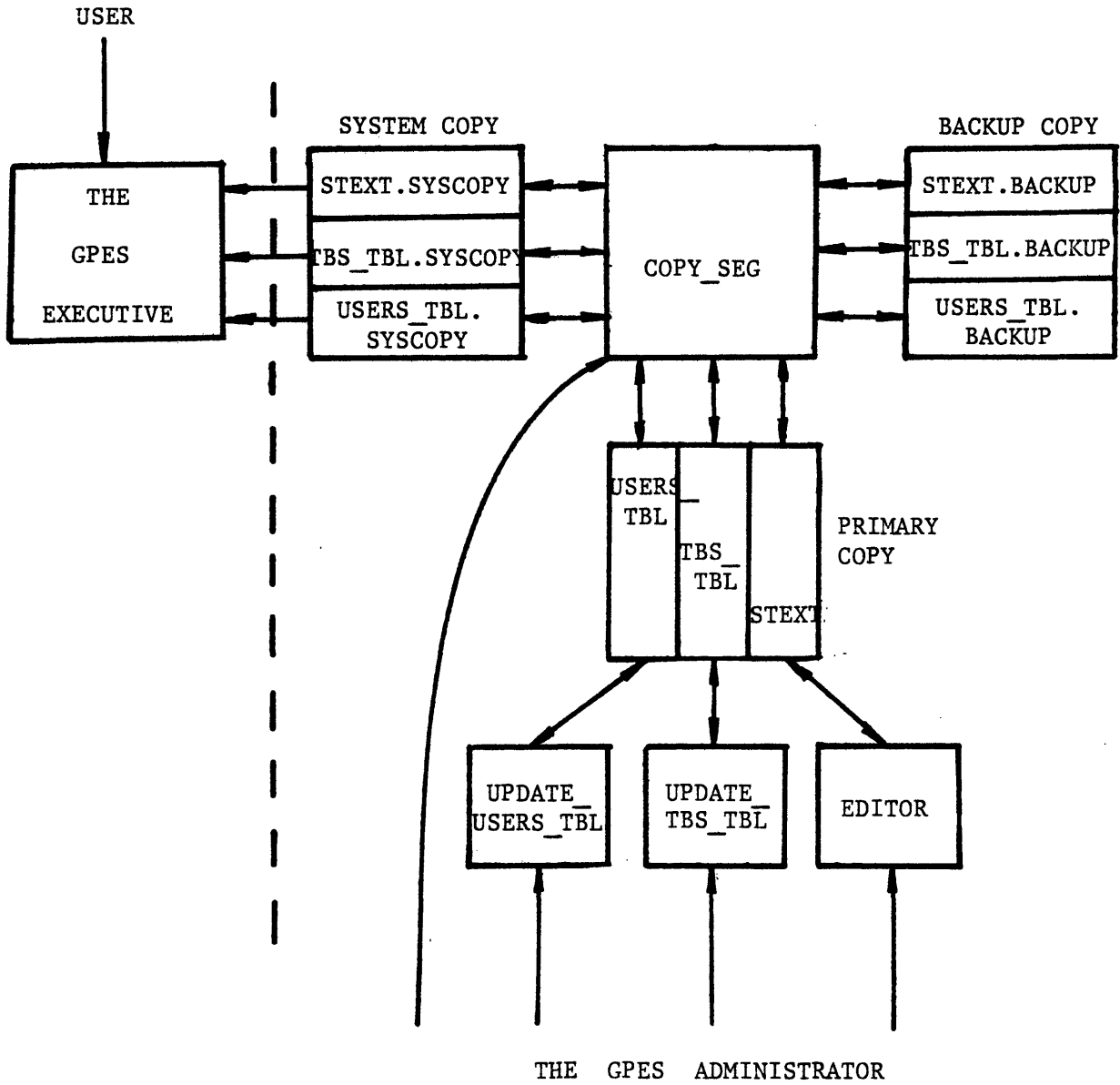


FIGURE 8.6 INFORMATION FLOW REGARDING GPES FILES

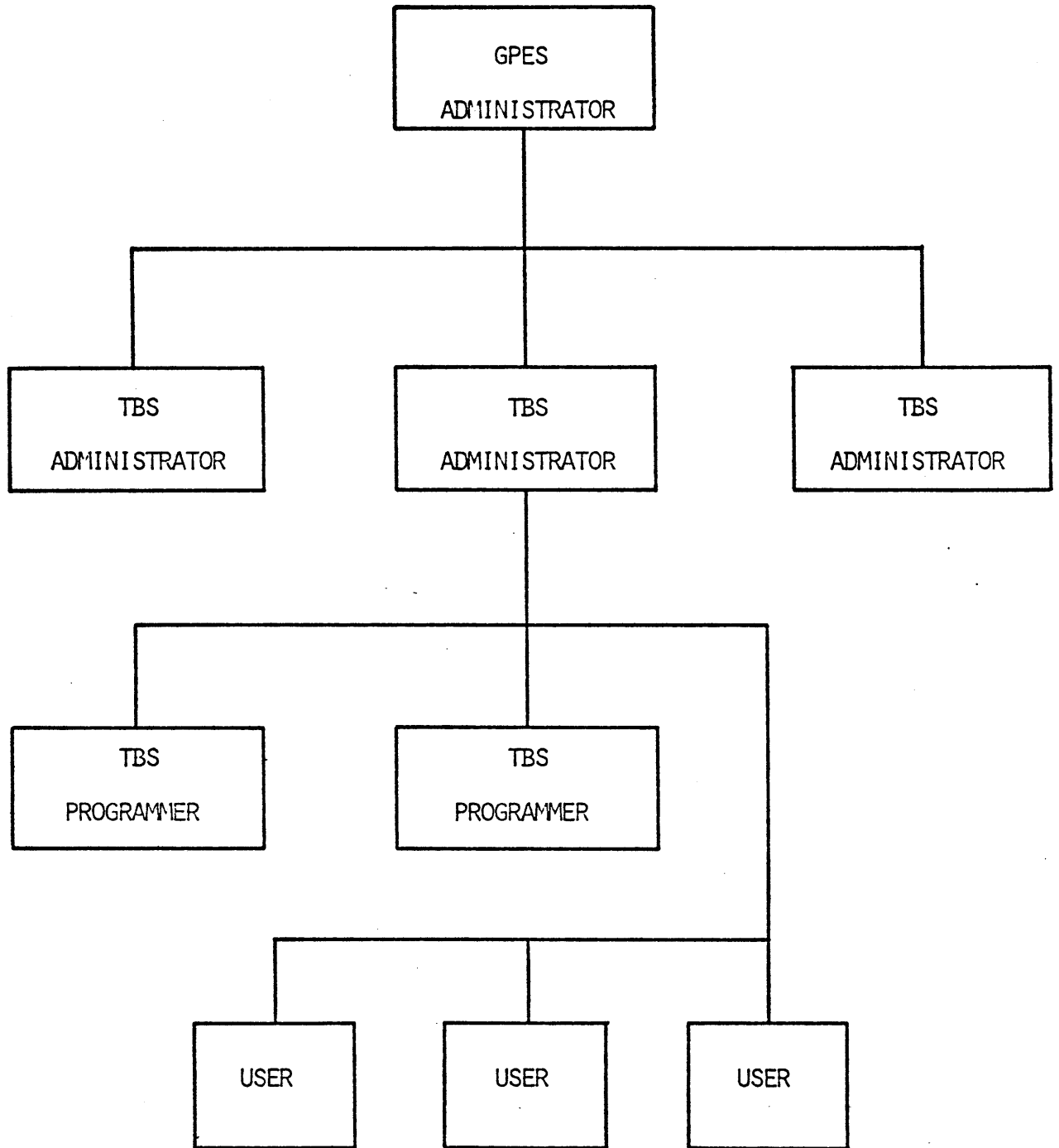


FIGURE 8.7 THE HIERARCHICAL STRUCTURE OF GPES USAGE

TABLE 8.4 GPES PROGRAMS USED BY EACH GROUP

Group	Program	Function	Language Used to Communicate with the Program
1) GPES Administrator	update_tbs_tbl	To create and update the TBS table	A set of commands
	update_users_tbl	To create and update the users table	A set of commands
	any editor	To create and update the GPES text file	Editor commands
	copy_seg	To copy a segment into another one	
2) TBS Administrators	update_tdb	To create and update the template data base	Template Definition Language (TDL)
	any editor	To create and update the TBS text file	Editor commands
	copy_tdb	To copy the template data base	
	delete_tdb	To delete a copy of a template data base	
	gaccess_tdb	To give users access to TDB	
	taccess_tdb	To deny users access to TDB	
	gaccess_sub	(to be developed by TBS administrators) To give users access to TBS programs	
	taccess_sub	(to be developed by TBS administrators) To deny users access to TBS programs	
3) TBS Programmers	service routines	To perform various tasks for a TBS program	
	compilers, etc.	To develop TBS programs	Procedural languages such as Fortran, PL/1, etc.
4) USERS	GPES executive	To solve users problems	Process Engineering Language (PEL)

CHAPTER 9

THE GPES EXECUTIVE

The Executive is an interactive program which executes PEL commands. It is the backbone of the system. In essence the executive is a table-driven interpreter, the tables being the template data bases and GPES files.

The program processes one user command at a time. Before the program can process user commands, the working environment, within which the commands must operate, must be prepared. Therefore, the program's task consists of a) environment preparation and b) command processing.

The former consists of the following three phases:

- 1) System initialization phase. As shown in Figure 9.1, the user is admitted to the system and a working area is established for him. The token table which will be described later is also allocated.
- 2) TBS initialization or TBS attachment phase. In this phase the user's desired TBS is initiated and attached to the system as shown in Figure 9.2.
- 3) Process initialization phase. As shown in Figure 9.3, the Process Directory is initialized and the data structures for "Property Estimation Methods in Use" and "Component Directory" are created.

The command processing loop consists of the four following phases:

- 1) Lexical analysis. In this phase a command (a string of characters) is read and decomposed into recognizable symbols called tokens. The tokens are placed into the token table.
- 2) Command recognition phase. In this phase the command is recognized by its first two tokens.

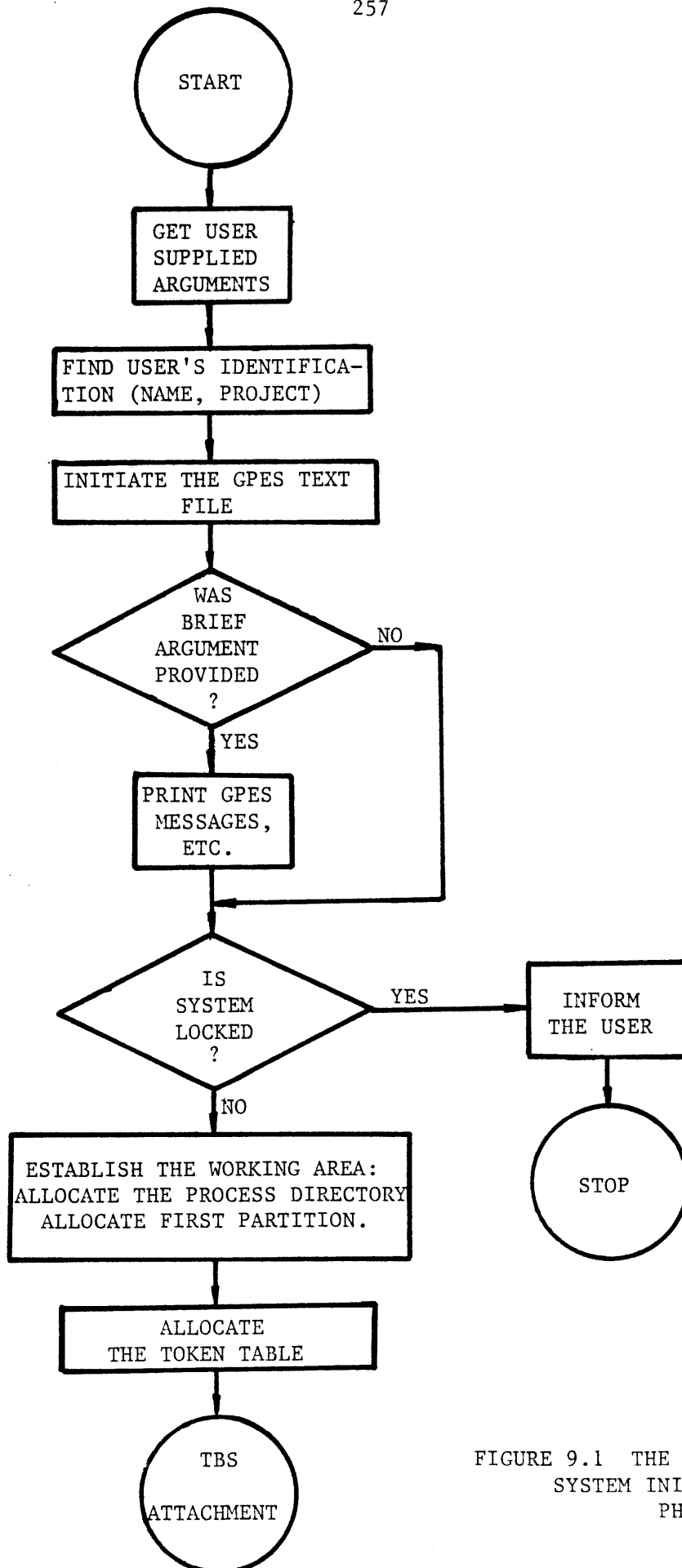


FIGURE 9.1 THE FLOW CHART FOR SYSTEM INITIALIZATION PHASE

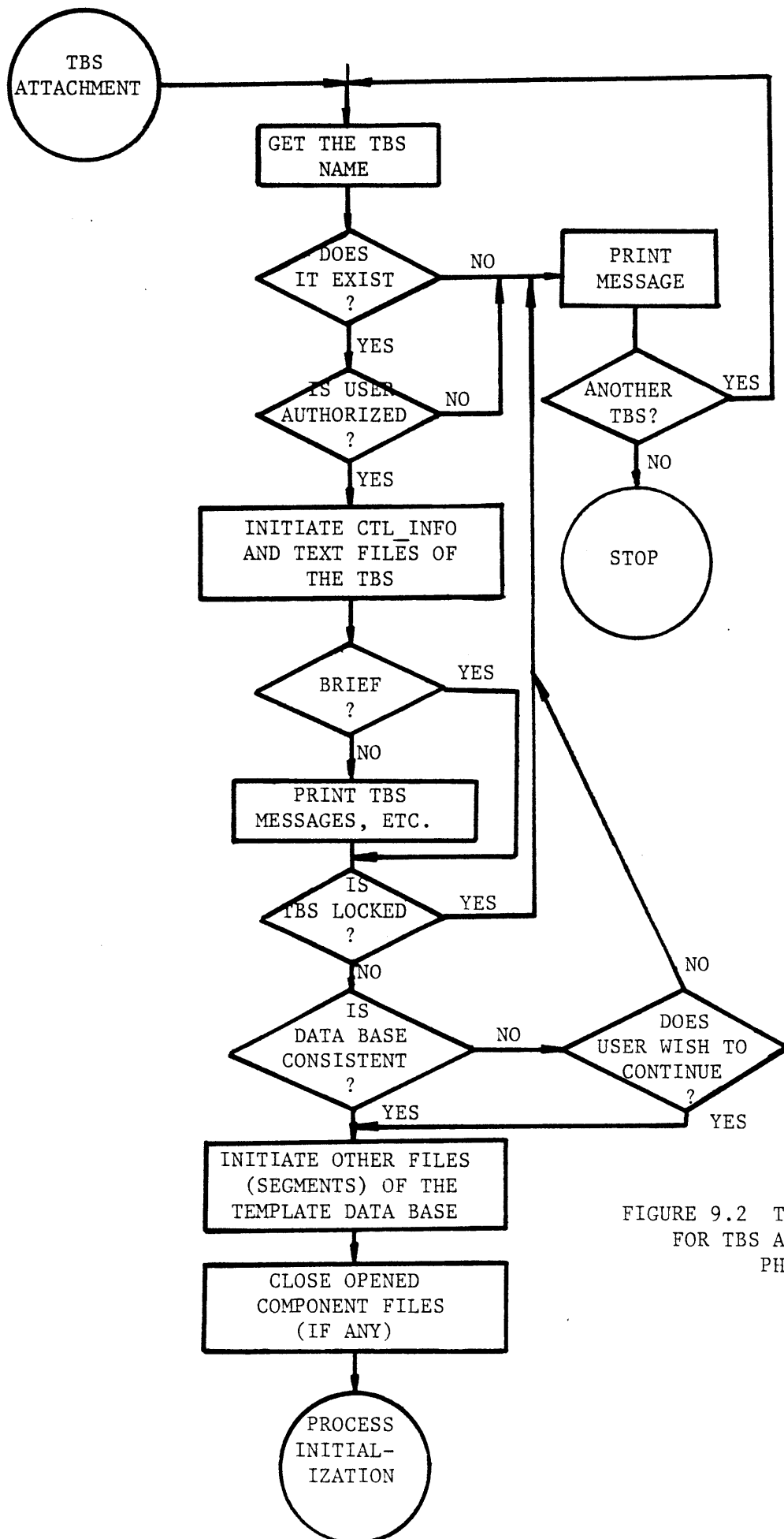


FIGURE 9.2 THE FLOW CHART FOR TBS ATTACHMENT PHASE

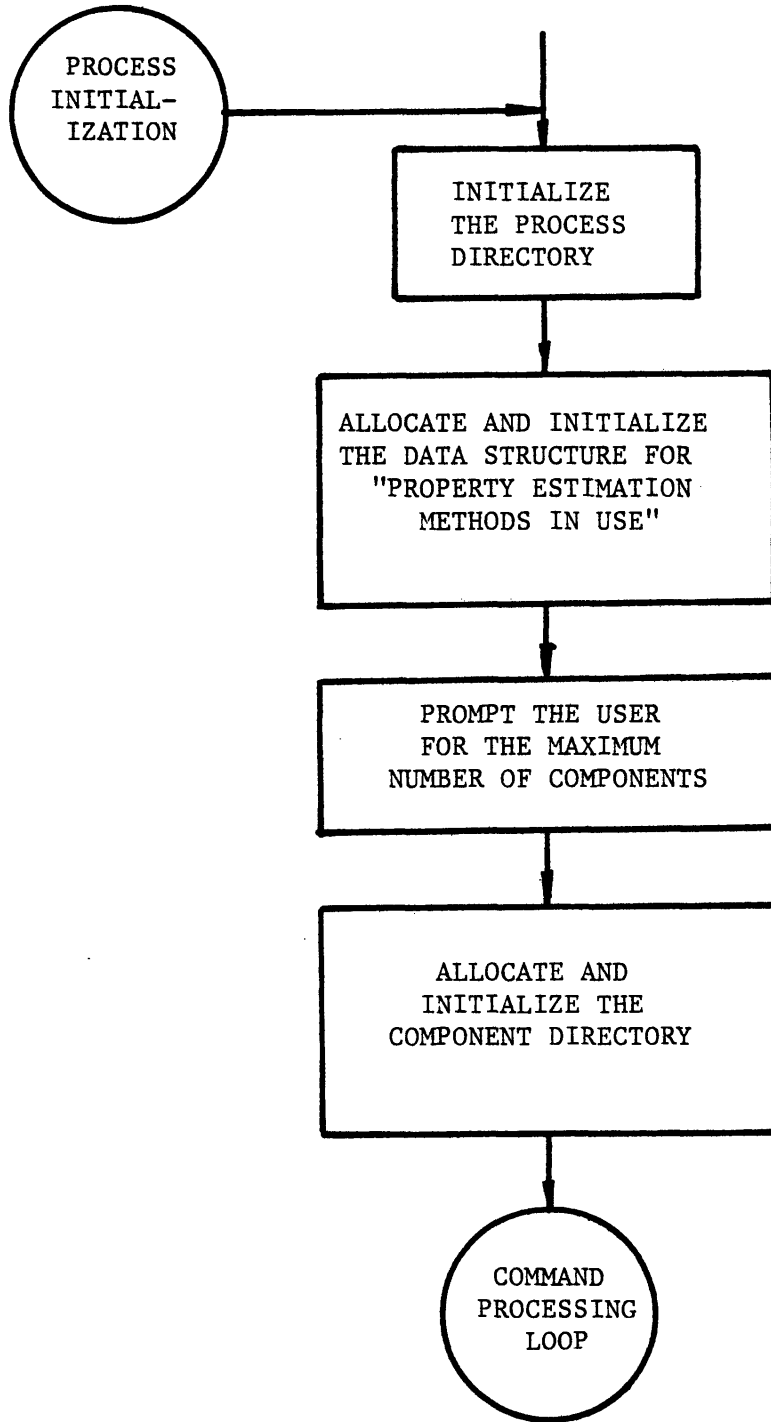


FIGURE 9.3 THE FLOW CHART FOR PROCESS INITIALIZATION PHASE

- 3) Syntax analysis and interpretation phase. Once a command is recognized it will be checked for its syntax and simultaneously its meaning will be interpreted. For some commands an intermediate form representing the semantics of the command will be constructed.
- 4) Execution phase. In this phase the command is executed. The execution phase being the last step in the command processing loop, ensures that the command is only executed when it is syntactically and semantically correct.

The program processes the user's commands one at a time. Therefore, control remains in the command processing loop except for the following commands:

- a) end command which terminates the program.
- b) clear command which clears the working area and transfers control to the process initialization phase.
- c) leave command which transfers control to the TBS attachment phase where a new TBS is attached.
- d) escape command which transfers control to the Multics command level where the user may execute any number of Multics commands. To return to PEL command processing loop the user should enter a blank line.

9.1 The GPES Executive Structure

The executive consists of a main program called "pel" and a number of other external programs. The structures of the "pel" program and of each of these external programs are modular. This is expected to facilitate the maintenance of the system, which otherwise would have been

difficult considering its size. The "pel" program consists of a main section, over 60 internal routines, and a "begin block" for each command. The "begin block" is a PL/1 terminology which refers to a group of PL/1 statements. A begin block is similar to an internal routine with the exception of the way control is passed to it. Control is passed to a begin block by a goto statement or through normal sequential execution. The layout of the "pel" program is shown in Figure 9.4. The main section of the "pel" program monitors the execution of other routines and begin blocks. It calls upon the external program "lexical" to perform the first phase of the command processing task. Then it performs the command recognition phase. Once the command is recognized it passes control to the associated begin block of the command. The begin block of a command performs the syntax checking and interpretation of the command. These begin blocks in turn will call upon various internal routines to perform various functions. For some commands an intermediate form representing the semantics of the command will be constructed to be passed to the execution phase. The execution of a command is either performed in the same begin block or an internal or external procedure is invoked to perform the task.

In addition to "pel" program, the GPES executive consists of the following external programs:

- 1) The "tbs_validation" and "tbs_access" programs. These two programs are called by the "pel" program in TBS attachment phase to get information about a TBS. In fact, both tbs_validation and tbs_access are entries of a single program which also has two other entries (update_tbs_tbl, update_users_tbl) by which the GPES administrator updates the TBS

GLOBAL DECLARATIONS	
VARIOUS INTERNAL ROUTINES	.
	.
	.
MAIN SECTION	SYSTEM INITIALIZATION PHASE
	TBS ATTACHMENT PHASE
	PROCESS INITIALIZATION PHASE
	READ: CALL LEXICAL(0); LEXICAL ANALYSIS PHASE
	COMMAND RECOGNITION PHASE
	GO TO APPROPRIATE BEGIN BLOCK
BEGIN BLOCKS (ONE FOR EACH COMMAND)	.
	GO TO READ;
	.
	GO TO READ;
	.
	.
	.
BEGIN BLOCK FOR PRINTING ERROR MESSAGES	
GO TO READ;	

FIGURE 9.4 THE LAYOUT OF THE "pel" PROGRAM

Table and Users Table, as described in Chapter 8. By referring to the TBS table the tbs_validation program verifies if a given TBS is in existence or operational and returns the address of the associated template data base. The tbs_access program verifies the access rights of the user regarding the specified TBS by referring to the Users Table.

- 2) The "lexical" program. The main purpose of this program is the lexical analysis as will be described later.
- 3) The "cdbsys" program. This program is called for the execution of commands related to component files. Its main function is the manipulation and data management of component files.
- 4) The "pdbsys" program. This program is called for the execution of commands related to process files. Its main function is the manipulation and data management of process files.
- 5) The "print_temp" program. This program which is also used by "update_tdb" program prints information about the templates. It is called in the execution phase of "printt" and "listt" commands.
- 6) The service routines getline, get_response, getin, and getrn. These routines which are part of the package of service routines developed for TBS

programmers are also used by "pel" and other programs. The functions of these programs are as follows:

getline	To receive a line of characters from the user.
get_response	To receive a yes or no answer from the user.
getin	To receive an integer number from the user.
getrn	To receive a real number from the user.

9.2 Lexical Analysis Phase

The action of parsing the command (a string of characters) into the proper syntactic classes is known as Lexical Analysis. The operational details for this step involve conceptually simple string processing techniques. The input command (a string of characters) is scanned sequentially. The basic elements or tokens are delimited by blanks and operators, and thereby recognized as identifiers, special identifiers, literals, or terminal symbols. The basic elements (tokens) are placed into a table called a token table. The token table is shown in Figure 9.5. It contains one command at a time. The table grows or shrinks automatically to accommodate any size command. Each entry of the table contains a token. The "symbol" contains the character representation of the token and "type" indicates the syntactic class of the token. Each PEL command terminates with a ";". Hence, the last entry of the token table is always a ";". The types of each class of tokens are listed in Table 9.1. The comments are discarded in the Lexical Phase, since they have no effect on the execution of the command.

TOKEN TABLE

NUMBER OF TOKENS = n		
1	SYMBOL	TYPE
2		
n	;	1

FIGURE 9.5 THE TOKEN TABLE

Table 9.1 Token Class Types

I) <u>Terminal Symbols</u>	<u>Type</u>
;	1
:	2
,	3
' (1)	4
)	5
/	6
&	7
	8
=	10
-	11
+	12
*	13
>	14
<	15
(16
^=	18
**	19
>=	20
<=	21
II) <u>Numbers</u>	24
III) <u>Identifiers</u>	25

Table 9.1 continued

IV) Composite Identifiers (2)

First element of a unit qualified variable	27
First element of a component qualified variable	28
First element of a function qualified variable	29
First element of a stream qualified variable	30
First element of a flow qualified variable	31
Other elements of a qualified variable	25

Notes:

1. The dimension field (indicating the units of measurement) is always enclosed in a pair of apostrophes. Therefore, it consists of three tokens: two apostrophes and the dimension field itself. For simplicity only the latter is placed in the token table with a type of 4.
2. The separating points of a composite identifier are not placed in the token table.

The external program "lexical" is called to perform the lexical analysis. The program prompts the user for a line of a command and then performs lexical analysis on that line. The program detects lexical errors (e.g. improper identifier, invalid composite identifiers, etc.) and prompts the user to reenter the line in error. Then the program asks for a new line and repeats the above process until the command terminator ";" is received. Note that there is no limit on the size of a PEL command. A PEL command may consist of any number of lines and each line may be up to 262 characters. The program also prints the contents of the token table if the profile parameter dflag (debugging flag) is equal to three, as demonstrated in Figure 9.6. The lexical program has also two other modes of operation:

- 1) In this mode of operation the program reads a line of characters (a Multics command) and passes it to the Multics command processor to be executed. This process is repeated until the input command is a blank line when the program returns control to the caller (pel program). The lexical is called with this mode of operation in response to the user's escape command.
- 2) This mode of operation is like the original mode of operation with one exception. The exception is that the program returns to the caller after processing only one line of input. The program is called with this mode of operation in execution phase of read and reada commands.

9.3 Command Recognition Phase

In this phase the first or the first and second tokens of the command are examined to recognize the command. Each command begins with a keyword called a command verb and may be followed by another keyword called the command object. In this phase a verb-code and object-code are associated with the command and for further analysis control is passed to the

FIGURE 9.6 AN EXAMPLE TO DEMONSTRATE THE FUNCTION OF LEXICAL ANALYSIS PHASE

```

**COMMAND :          $---A REQUEST OF PRINTOUT OF THE TOKEN TABLE AMONG OTHER INFORMATION---$
**continue:profile dflas=3;
**COMMAND :          $---THE FOLLOWING COMMAND CONTAINS ALL TYPES OF TOKENS---$
**continue:specify variables(s.feed.p0.t=u.a.%p1+fn.h.a**2'f' , x=h(s,-c.co.tc*4))
**continue:if(y<1/4 & f.feed.p0.co.x<=y & d>.4 ! y>=12.3e-4 & y^=15);

 1 specify          25
 2 variables        25
 3 (                16
 4 s                30
 5 feed            25
 6 p0              25
 7 t               25
 8 =               10
 9 u               27
10 a               25
11 %p1             25
12 +               12
13 fn              29
14 h               25
15 a               25
16 **              19
17 2               24
18 f                4
19 ,                3
20 x               25
21 =               10
22 h               25
23 (                16
24 s               25
25 ,                3
26 -               11
27 c               28
28 co              25
29 tc              25
30 *               13
31 4               24
32 )                5
33 )                5
34 if              25
35 (                16
36 y               25
37 <               15
38 1               24
39 /                6
40 4               24
41 &                7
42 †               31
43 feed            25
44 p0              25
45 co              25
46 x               25
47 <=              21
48 y               25
49 %                7
50 d               25
51 >               14
52 .4              24
53 !                8
54 y               25
55 >=              20
56 12.3e-4         24
57 %                7
58 y               25
59 ^=              18
60 15              24
61 )                5
62 †                1
***ERROR*** s 30 'feed' is an unknown stream .
*command ignored.
**COMMAND :

```

associated begin block of the command. The verb-codes and object-codes are integer numbers representing command verbs, and command objects respectively. Table 9.2 lists the object codes. The data structure "command object flag" indicates the acceptable objects for each command. The data structure has an entry for each command which consists of seven bits, one for each object. Each bit is "on" or "off" depending on whether the corresponding object is acceptable for the command or not, as shown in Table 9.3. The flowchart for command recognition phase is shown in Figure 9.7.

9.4 Syntax Analysis and Interpretation Phase

The token table is scanned to check the syntax and interpret the meaning of the command. The global variable "start" is used to point to the current token under consideration. For each command there is a begin block performing syntax checking and interpretation. Of course, these blocks call upon various internal routines to perform various functions. For some commands intermediate forms representing the semantics of the commands will be constructed to be passed to the execution phase. This is required for commands that appear inside a repeat-loop group and must therefore be executed repetitively. The PEL commands can be classified into three groups according to their appearance in a repeat-loop group:

- 1) Those that are not permitted in a repeat-loop. The commands in this group are as follows: let, leta, delete, and unspecify. Note that let and leta commands may implicitly unspecify a parameter.
- 2) Those that if appearing inside a repeat-loop only will be executed once, independent of the condition of the repeat-loop. Some of these commands even terminate one or all of the repeat-loops (e.g. stop, clear, leave, and load process).

Table 9.2 Command Object Codes

<u>Object</u>	<u>Object Code</u>
unit	1
component	2
function	3
stream	4
flow	5
variable	6
process	7

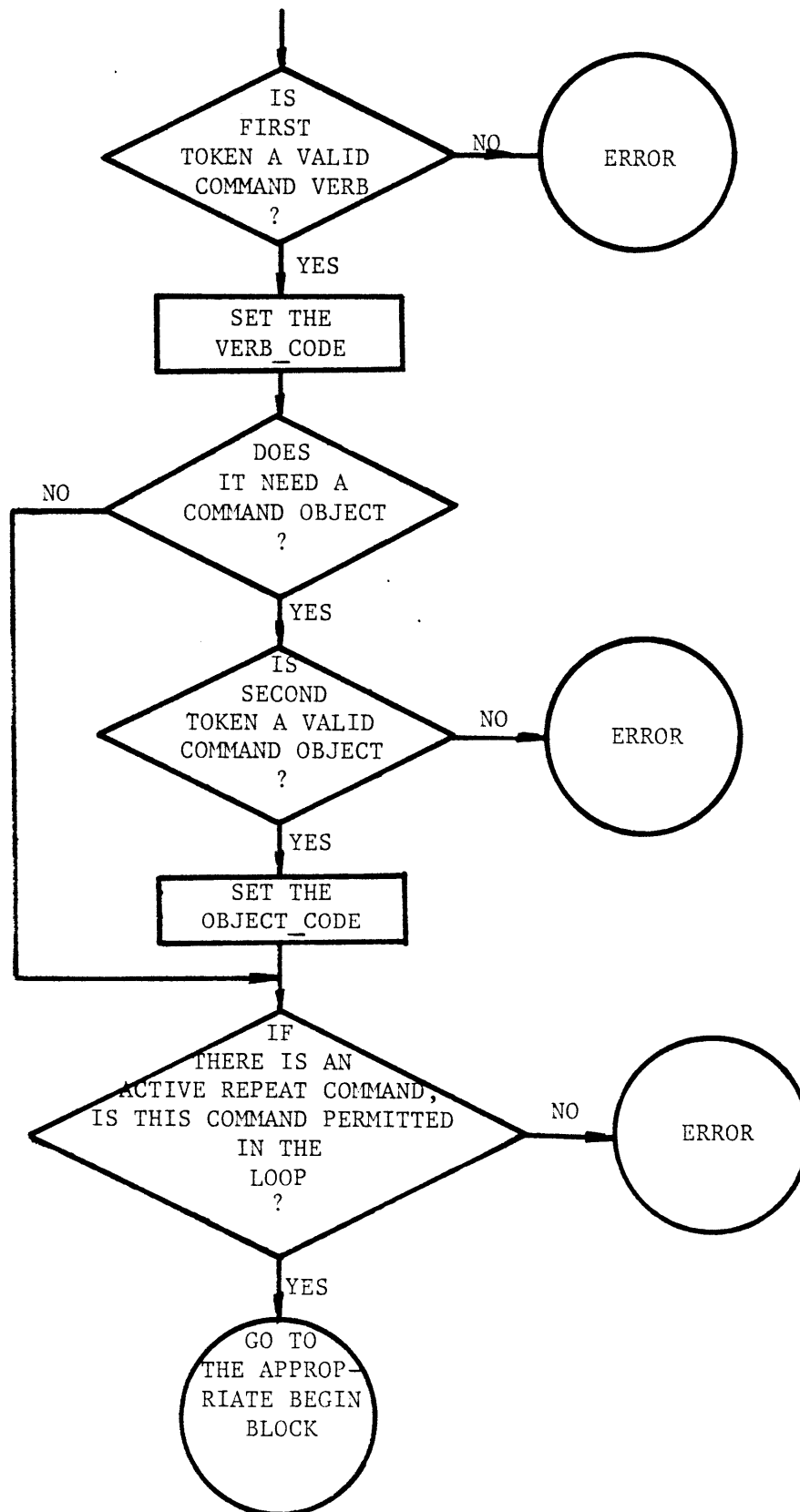
Table 9.3 Command Verb Codes and Object Flags

Verb_code		(unit)	(component)	(function)	(stream)	(flow)	(variable)	(process)
		(1)	(2)	(3)	OBJECT FLAGS		(6)	(7)
					(4)	(5)		
1	repeat	0	0	0	0	0	0	0
2	loop	0	0	0	0	0	0	0
3	assume	1	1	1	1	1	1	0
4	specify	1	1	1	1	1	1	0
5	calculate	1	1	1	1	0	0	0
6	print	1	1	1	1	1	1	1
7	leta	1	1	1	1	1	0	0
8	let	1	1	1	1	1	0	0
9	delete	1	1	1	1	1	0	0
10	unspecify	1	1	1	1	1	1	0
11	disconnect	1	0	0	0	0	0	0
12	create	1	1	1	1	1	0	0
13	connect	1	0	0	0	0	0	0
14	list	1	1	1	1	0	1	0
15	printt	0	0	0	0	0	0	0
16	open	0	1	0	0	0	0	1
17	save	0	1	0	0	0	0	1
18	load	0	1	0	0	0	0	1
19	deletef	0	1	0	0	0	0	1
20	listf	0	1	0	0	0	0	1

FIGURE 9.3 CONTINUED

		(1)	(2)	(3)	(4)	(5)	(6)	(7)
21	terminate	0	1	0	0	0	0	1
22	close	0	1	0	0	0	0	1
23	printf	0	1	0	0	0	0	0
24	copy	0	1	0	0	0	0	0
25	include	0	1	0	0	0	0	0
26	use	0	0	0	0	0	0	0
27	end	0	0	0	0	0	0	0
28	stop	0	0	0	0	0	0	0
29	clear	0	0	0	0	0	0	0
30	profile	0	0	0	0	0	0	0
31	news	0	0	0	0	0	0	0
32	help	0	0	0	0	0	0	0
33	listt	1	1	1	1	0	0	0
34	bugs	0	0	0	0	0	0	0
35	leave	0	0	0	0	0	0	0
36	continue	0	0	0	0	0	0	0
37	escape	0	0	0	0	0	0	0
38	reada	1	1	1	1	1	1	0
39	read	1	1	1	1	1	1	0

FIGURE 9.7 THE FLOWCHART FOR COMMAND RECOGNITION PHASE



3) Those that when appearing inside a repeat-loop will be executed repeatedly depending on the condition of the repeat-loop and their "if-clauses". The following commands fall in this group: repeat, loop, assume, specify, read, reada, calculate, and print.

The syntax of most of the commands falling in the first two groups is rather simple. Consequently the tasks of syntax analysis and interpretation of these commands are straightforward. Once these commands are analyzed and executed they are no longer needed. On the other hand the syntax of commands falling in the third group is usually more sophisticated and also they have to be executed repeatedly if appearing inside a repeat-loop command. Therefore, the discussion in this section will be limited to these commands.

An intermediate form for each of these commands will be constructed. The intermediate form affords two advantages:

- 1) Simplifying the task of the execution phase by providing all the relevant information for execution.
- 2) Representing the command in a repeat-loop.

Arithmetic expressions may appear in these commands and other commands. Therefore, before describing the intermediate forms of these commands the intermediate form of arithmetic expressions is described.

9.4.1 The Intermediate Form of Arithmetic Expressions

The intermediate form of an arithmetic expression is placed in the data structure "parsed matrix" shown in Figure 9.8.

In this matrix operations of the expression are listed sequentially in the order that they would be executed to evaluate the expression.

The parsed matrix data structure is also used to represent the mathematical expression of the user defined functions as discussed in Chapter 5.

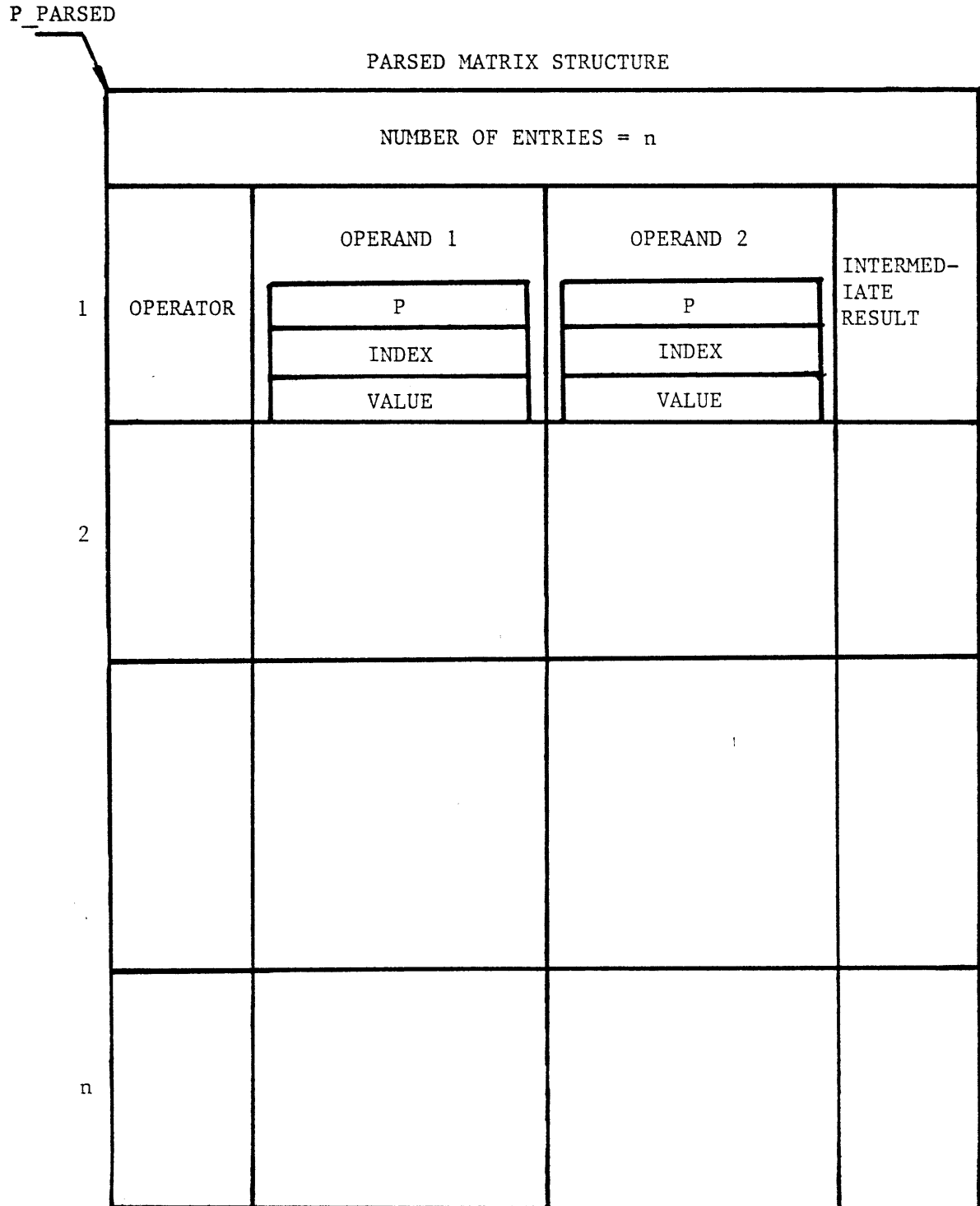


FIGURE 9.8 THE PARSED MATRIX STRUCTURE

Each matrix entry has one operator, two operands, and a location to store the result at the time of evaluation. The operator indicates the type of operation to be performed on the operands. The different types of operations in the parsed matrix are shown in the Table 9.4. The operator ",", indicates that the two operands are part of an argument list of a function reference as will be demonstrated in the example that follows later in this section. The end of matrix operation which is the last entry of the matrix converts the result of the expression to the standard units of measurement. For this entry operand 2 represents the result of the expression before the conversion and operand 1 contains the "B" coefficient. "A" coefficient is placed in the location of the intermediate result. The result of operation and consequently the result of the expression is $A+B$ (operand 2). Coefficients "A" and "B" are always supplied; even when the expression represents the value of a dimensionless variable, "A" of zero and "B" of one are supplied.

An operand may be one of the following:

- 1) A number,
- 2) A simple variable,
- 3) A qualified variable,
- 4) The result of another entry (intermediate result),
- 5) A dummy argument which only appears for an expression representing (defining) a user defined function. In such an expression, simple variables and qualified variables are replaced with their values. Built-in constants are always replaced by their values.
- 6) A pre-defined function,
- 7) A user-defined function,
- 8) A built-in function.

TABLE 9.4 OPERATIONS IN THE PARSED MATRIX

Operation	Operator	Operand 1	Operand 2	Intermediate Result
Arithmetic Operation	Arithmetic Operator (+, -, *, /, or **)	Operand 1	Operand 2	
Comparison Operation	Comparison Operator (=, !=, >, >=, <, or <=)	Operand 1	Operand 2	
Boolean Operation	Boolean Operator (&, or)	Operand 1	Operand 2	
Negation (- prefix)	:		Operand	
Argument List (no operation)	,	Argument 1	Argument 2	
Function Reference	(or - (Function	Argument	
End of Matrix Operation	;	"B" Coefficient	Operand	"A" Coefficient

The last 3 types of operands may only appear as the first operand of a function reference operation.

The other types of operand may appear as the first or second operands for any operation (except as the first operand of a function reference operation). Therefore, the first operand of the entry having operators "(" or "-(" is always interpreted as a function.

Each operand is represented in the matrix by the following three items:

- 1) p - a pointer
- 2) index - an integer number
- 3) value - a real float number

Table 9.5 shows how each operand is uniquely represented by these three items. The index of a built-in function represents the function as it is listed in table 9.6.

The operators are not placed in the matrix as characters (e.g. +, *), but they are represented by integer numbers as shown in table 9.7.

An example of a parsed matrix is shown in Figure 9.9. For ease of reading the actual symbols are used as operators and operands. "Mi" denotes the matrix entry "i".

The internal routine "parser" performs the parsing of arithmetic expressions and it is called as follows:

```
call parser (fn_begin, dimension_type, terminate_code);
```

The global variable "start" points to the beginning of the expression in the token Table. For expressions representing a function the variable "fn_begin" points to the beginning of the dummy argument list in the token table for other expressions "fn_begin" is zero. The "dimension_type" indicates the dimension type of the variable or parameter to which the

TABLE 9.5 THE REPRESENTATION OF OPERANDS IN THE PARSED MATRIX

Operand	P	Index	Value
Number	Null	0	Number
Simple Variable	Pointer to the Variable's Structure	0	-
Qualified Variable	Pointer to the Parameters Structure	Parameter Number	-
Intermediate Result	Null	- Entry Number of The Matrix	-
Dummy Argument	Null	+ Argument Number	-
Pre-defined Function	Pointer to the Function's Structure	Entry Number of the Function Type in the Function Template directory	Number of Arguments
User-defined Function	Pointer to the Function's Structure	0	Number of Arguments
Built-in Function	Null	The index of the Built-in Function	Number of Arguments

TABLE 9.6 THE INDEX OF BUILT-IN FUNCTIONS IN THE PARSED MATRIX

<u>Index</u>	<u>BUILT-IN FUNCTION</u>
1	ABS
2	ACOS
3	ASIN
4	ATANH
5	COS
6	COSD
7	COSH
8	ERF
9	ERFC
10	EXP
11	LOG
12	LOG2
13	LOG10
14	SIGN
15	SIN
16	SIND
17	SINH
18	SQRT
19	TAN
20	TAND
21	TANH
22	ATAN(X1)
23	ATAND(X1)
24	MAX
25	MIN
26	MOD
27	ATAN(X1,X2)
28	ATAND(X1,X2)

TABLE 9.7 THE REPRESENTATION OF OPERATORS IN THE PARSED MATRIX

Operator	Symbolic Representation of the Operator	<u>Operation</u>
1	;	End of Matrix
2	:	Minus prefix (negation)
3	,	Argument List Construction
6	/	Division
7	&	"And" Operation
8		"Or" Operation
10	=	Comparison Operation
11	-	Subtraction
12	+	Addition
13	*	Multiplication
14	>	Comparison Operation
15	<	Comparison Operation
16	(Function Reference
17	-(- Function Reference
18	¬ =	Comparison Operation
19	**	Exponentiation
20	>=	Comparison Operation
21	<=	Comparison Operation

PARSED MATRIX FOR THE EXPRESSION:
 $10+2*(-4+F(X+Y,X*Y,Z)/A),$

	11		
1	:		4
2	+	X	Y
3	*	X	Y
4	,	M2	M3
5	,	M4	Z
6	(F	M5
7	/	M6	A
8	+	M1	M7
9	*	2	M8
10	+	10	M9
11	;	1	M10
			0

FIGURE 9.9 AN EXAMPLE OF A PARSED MATRIX

value of the expression will be assigned. It is zero for dimensionless parameters and variables. The dimension type allows the program to convert the value of expression to the standard units of measurement, if it has been provided in other units.

The "terminate_code" indicates the symbol designating the end of the expression. In PEL commands an expression is terminated either by "," (other than those separating the arguments of a function reference) or by a ")" (other than those matching the left parentheses of the expression). The "terminate_code" of zero indicates that the expression can be terminated either by "," or by ")". The terminate_code of one requires termination by ")" only.

The parser program scans the expression (token-table) on two passes. On the first pass the terminating token is recognized and replaced by a token which uniquely designates the end of expression ("," is replaced by the token of type 9, ")" is replaced by the token of type 1). The size of the expression and consequently the size of the parsed matrix are also found in this pass. On the second pass, the expression is parsed and placed in a parsed matrix structure. A modified operator precedence algorithm is used as shown in Figure 9.11. Associated with each operator there is a number called precedence. It indicates the priority of the operator as listed in Table 9.8. The data structure stack contains the tokens that are currently being worked on by the parser as shown in Figure 9.10. The tokens in the stack are represented by two integer numbers. One indicates the location of the token in the token table and the other indicates the type of the token. The reference to an entry of the matrix is by location equal to minus matrix entry number, and type equal to 40, if the matrix entry represents a value, or type equal to 41, if the matrix entry does not represent a value (when representing an argument list).

TABLE 9.8 THE PRECEDENCE TABLE

<u>Symbol</u>	<u>Type</u>	<u>Precedence</u>
;	1	0
:	2	0
,	3	3
'	4	0
)	5	1
/	6	8
&	7	5
	8	4
Not Used	9	0
=	10	6
-	11	7
+	12	7
*	13	8
>	14	6
<	15	6
(16	2
-(17	2
=	18	6
**	19	9
>=	20	6
<=	21	6

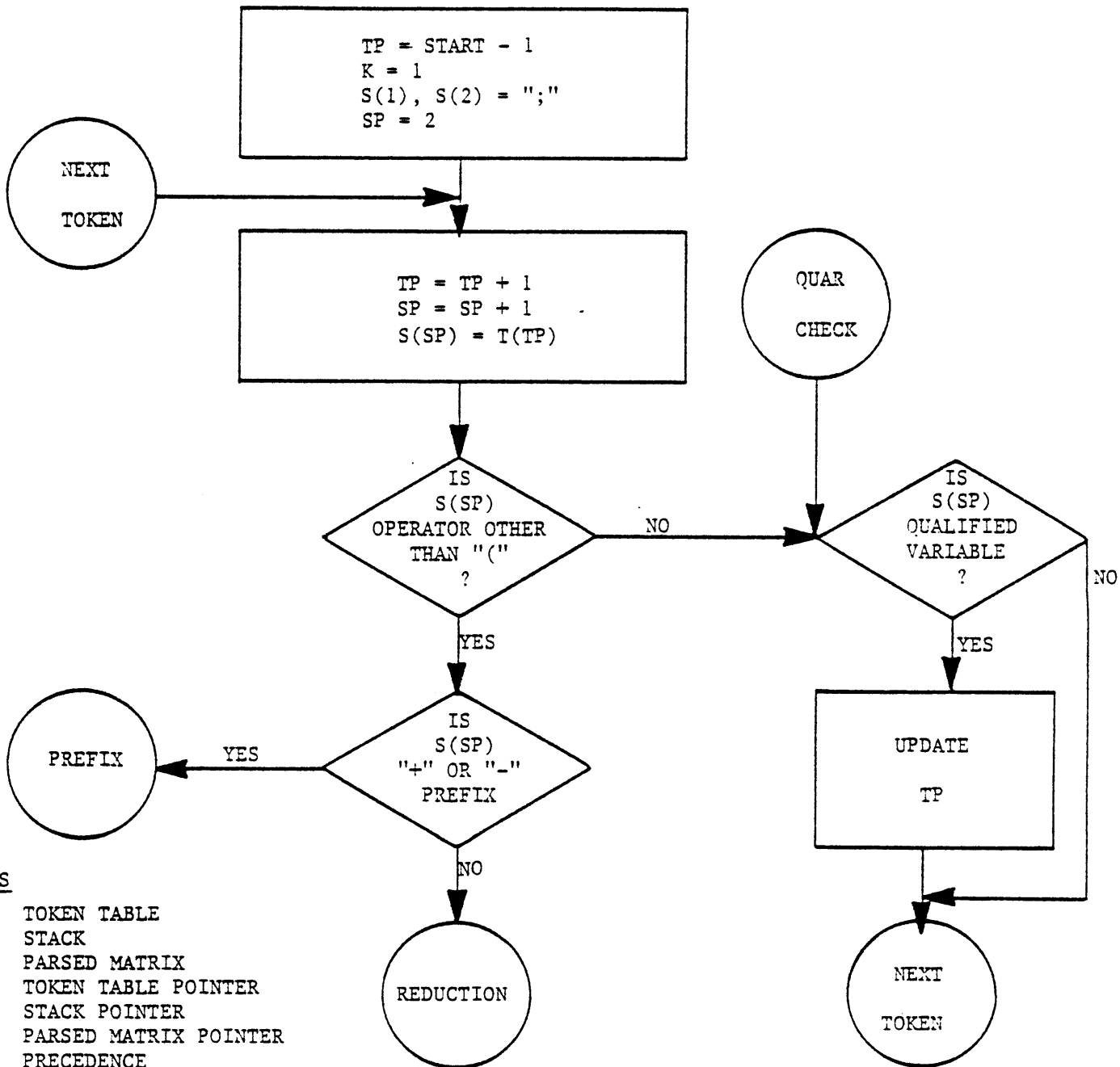
STACK

LOCATION	TYPE

ENTRY	LOCATION	TYPE
A TOKEN	INDEX OF THE TOKEN IN THE TOKEN TABLE	TYPE OF THE TOKEN
AN INTER- MEDI- ATE RESULT	MINUS INDEX OF THE INTER- MEDIATE RESULT (- ENTRY NUM- BER IN THE PARSED MATRIX)	40 (VALUE OR TYPE) 41 (ARGUMENT TYPE)

FIGURE 9.10 THE STACK USED IN PARSING THE EXPRESSIONS

FIGURE 9.11 THE PARSER ALGORITHM

NOTATIONS

T	TOKEN TABLE
S	STACK
M	PARSED MATRIX
TP	TOKEN TABLE POINTER
SP	STACK POINTER
K	PARSED MATRIX POINTER
P	PRECEDENCE
OPTYPE1, OPTYPE2	INDICATOR OF THE VALUE TYPE OF OPERANDS 1 AND 2
	OPTYPE = 40 VALUE TYPE
	OPTYPE = 41 ARGUMENT (LIST) TYPE

FIGURE 9.11 CONTINUED

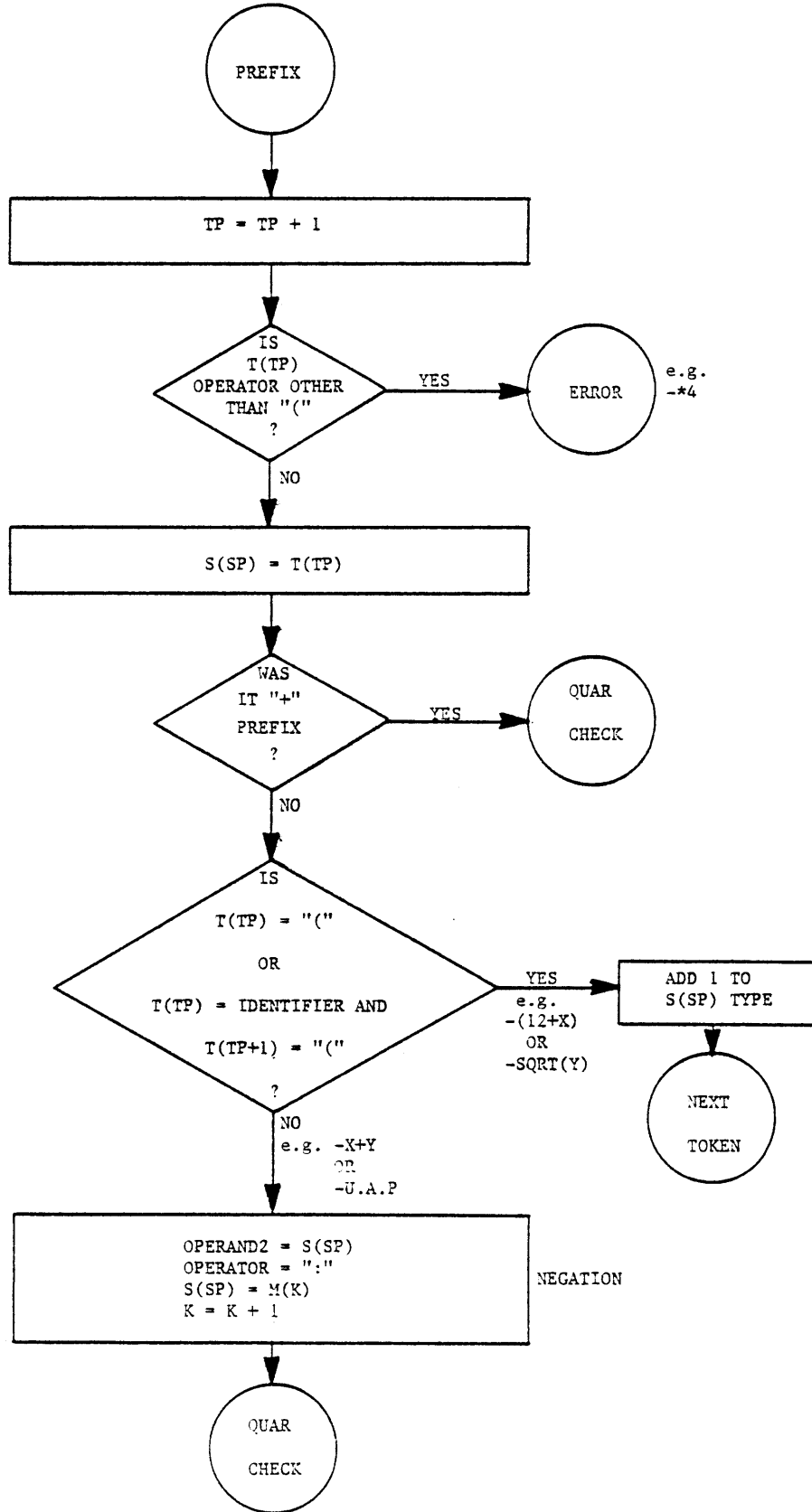


FIGURE 9.11 CONTINUED

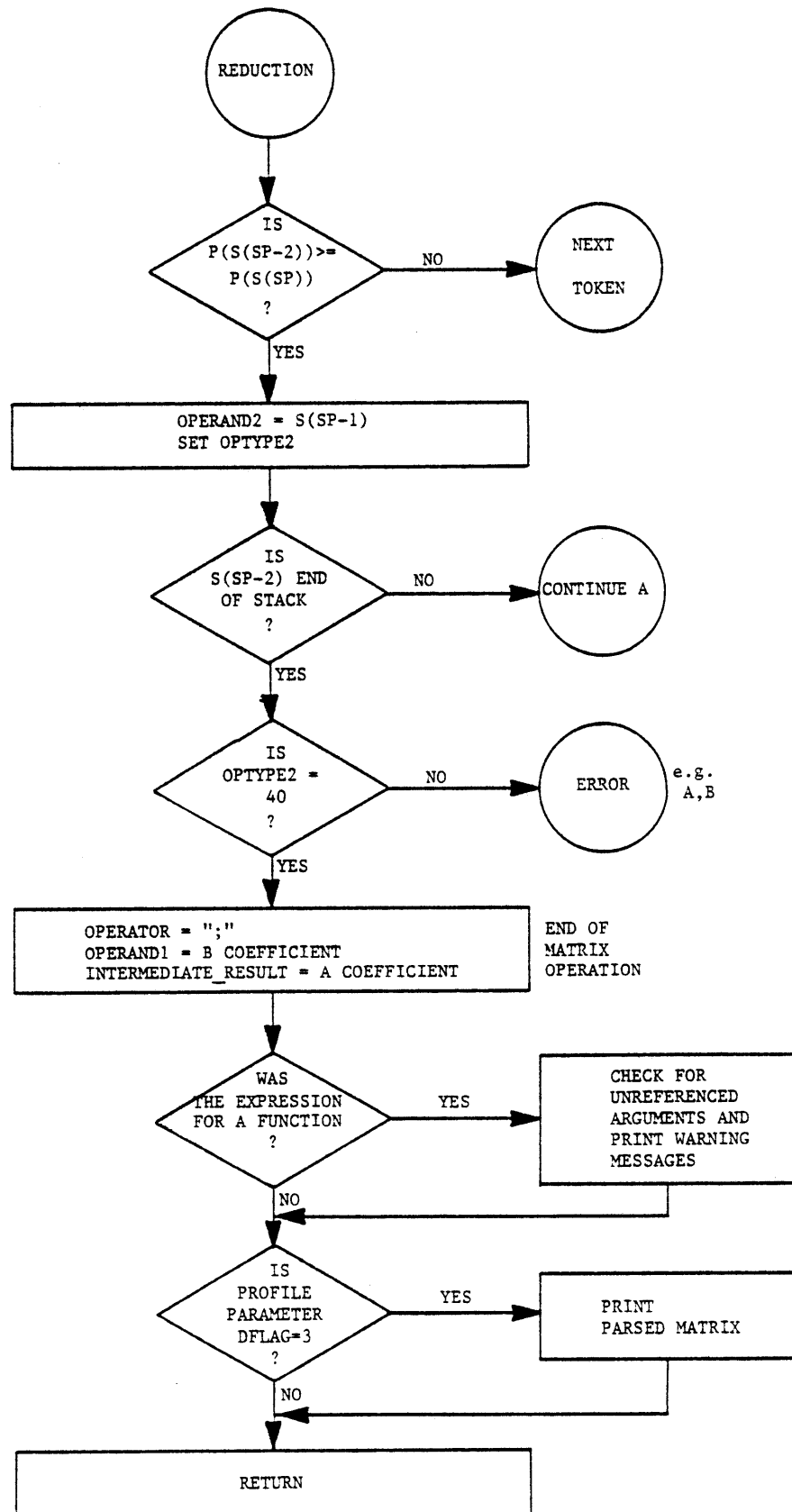


FIGURE 9.11 CONTINUED

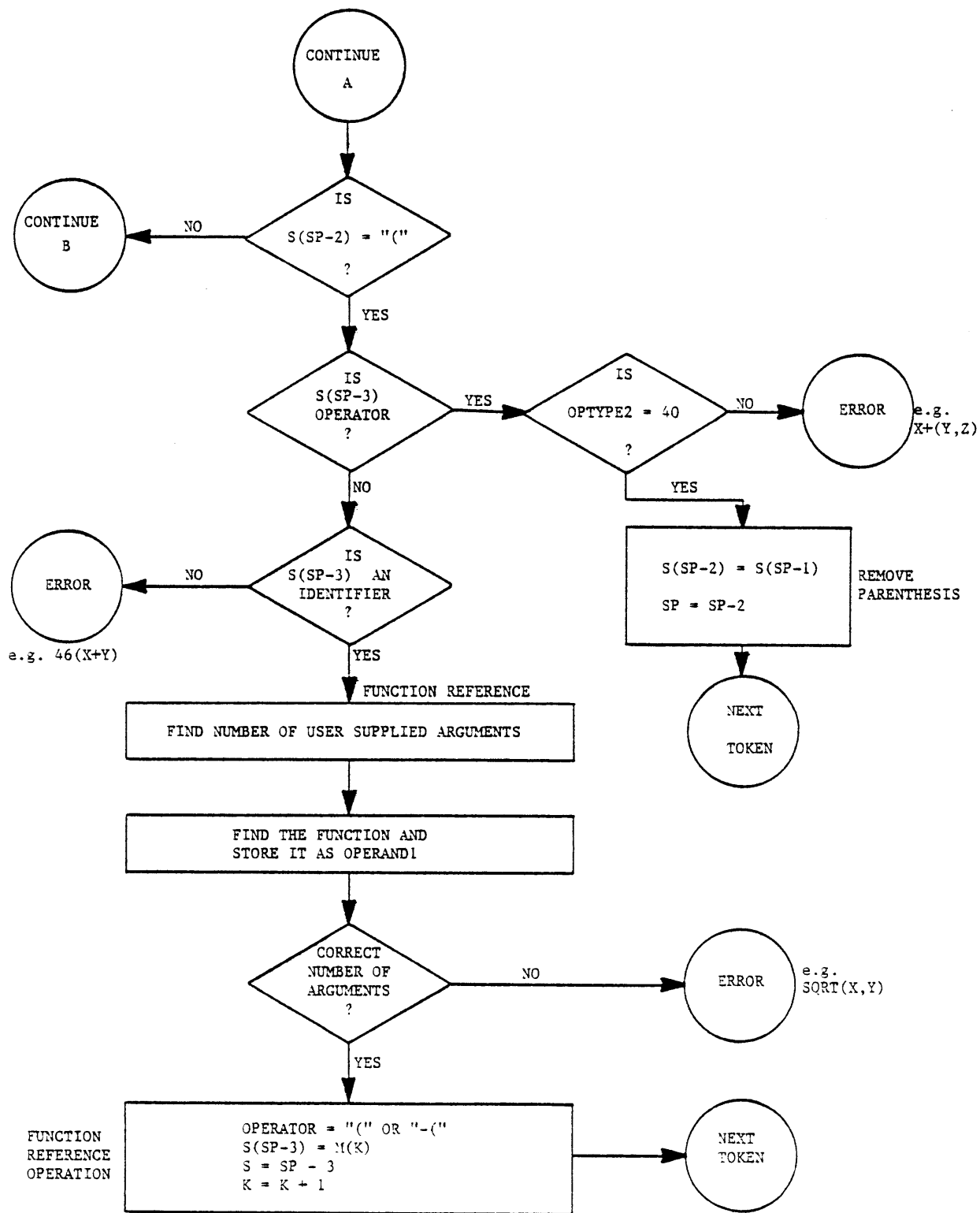
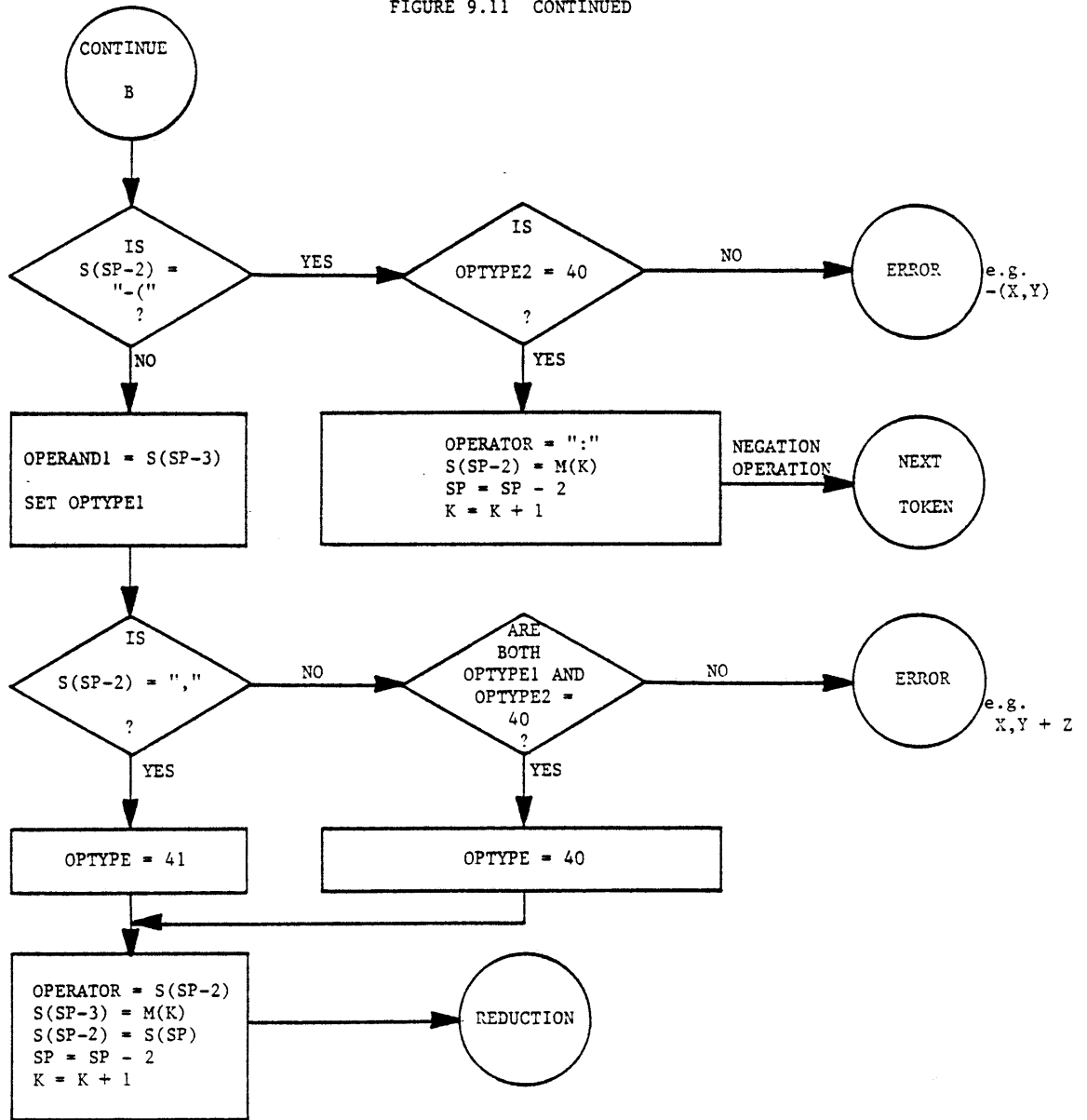


FIGURE 9.11 CONTINUED



9.4.2 Intermediate Forms of Commands

Various data structures are used to represent the commands in the third group. These data structures will be generally referred to as "control blocks" (cblock), indicating the fact that they are controlling the execution of the commands. If a command is not in a loop its control block will be deleted, once the command has been executed. But if a command is in a loop, its control block is linked to the control block of the previous command by the data structure shown in Figure 9.12. Therefore, there is one such data structure for each command in the loop. These data structures and associated control blocks will be deleted once the execution of the loop has been completed. The control blocks for various commands are listed in Table 9.9. Although the "unspecify" command does not belong to this group of commands, its syntax is very similar to the "read" command, and it is also represented by an intermediate form. Figures 9.13 to 9.20 show the control blocks and demonstrate their applications. The following notations are generally used in describing the control blocks:

p_parsed - pointer to the parsed matrix of an arithmetic expression.

pparm - pointer to a parameters structure.

parm_no - parameter number.

pvariable- pointer to a simple variable structure.

The notations used to describe the syntax of commands are defined in Appendix D. Although a command may not always be executed (if the condition in "if-clause" does not hold or if it is inside an unexecutable loop), its syntax analysis and interpretation phase (and consequently the construction of its control block) is always performed to detect its syntactic and semantic errors.

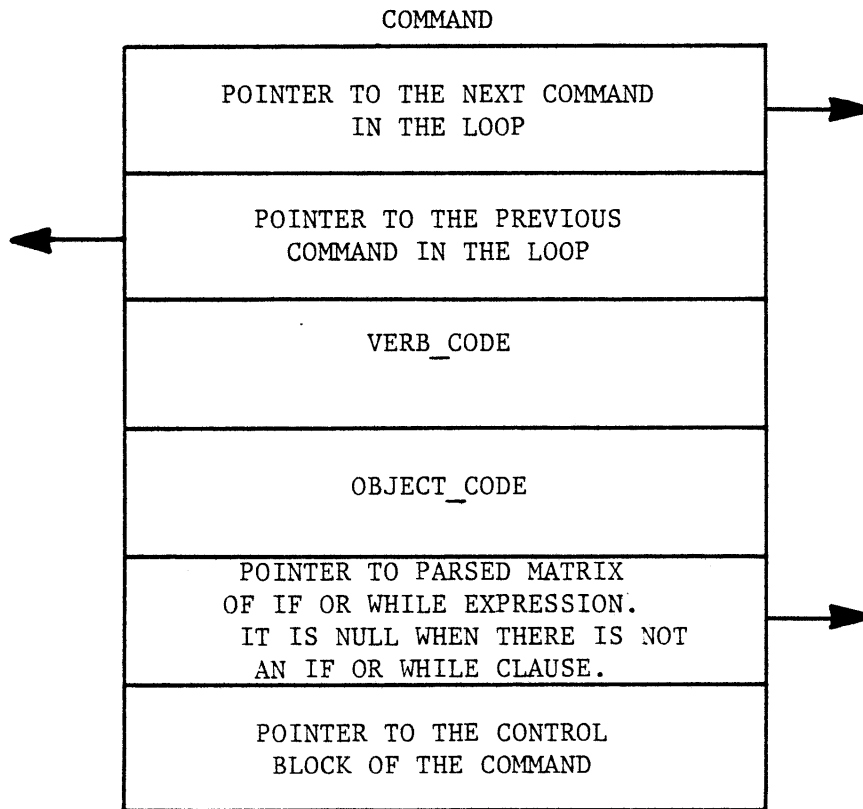


FIGURE 9.12 THE COMMAND HEADER STRUCTURE

TABLE 9.9 THE CONTROL BLOCKS OF VARIOUS COMMANDS

<u>Command</u>	<u>Control Block</u>
Repeat	CBlock
Loop	-
Calculate	CBlock5
Specify or assume variables	CBlock1
Specify or assume other objects	CBlock3
Print Variables	CBlock2
Print Components	CBlock6
Print other objects	CBlock4
Read, Reada, or Unspecify Variables	CBlock2
Read, Reada, or Unspecify other objects	CBlock7

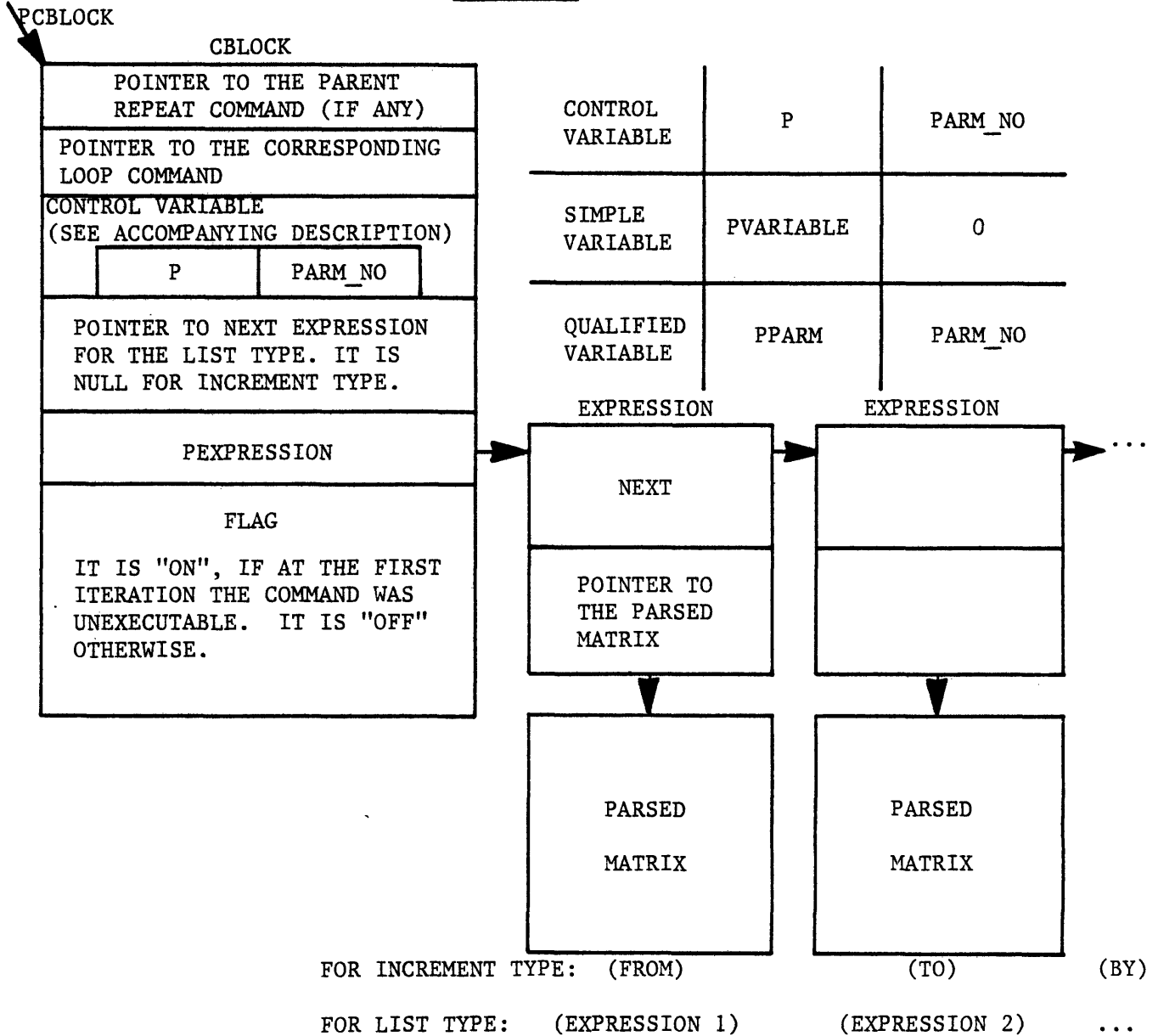
FIGURE 9.13 THE CBLOCK STRUCTURE

USED FOR: REPEAT COMMAND

SYNTAX:

INCREMENT TYPE: repeat for <variable> from(<expression>['dimension'])
to(<expression>['dimension'])[by(<expression>['dimension'])]
[while(<expression>)];

LIST TYPE: repeat for <variable>=<expression>['dimension']
[,<expression>['dimension']]*)[while(<expression>)];



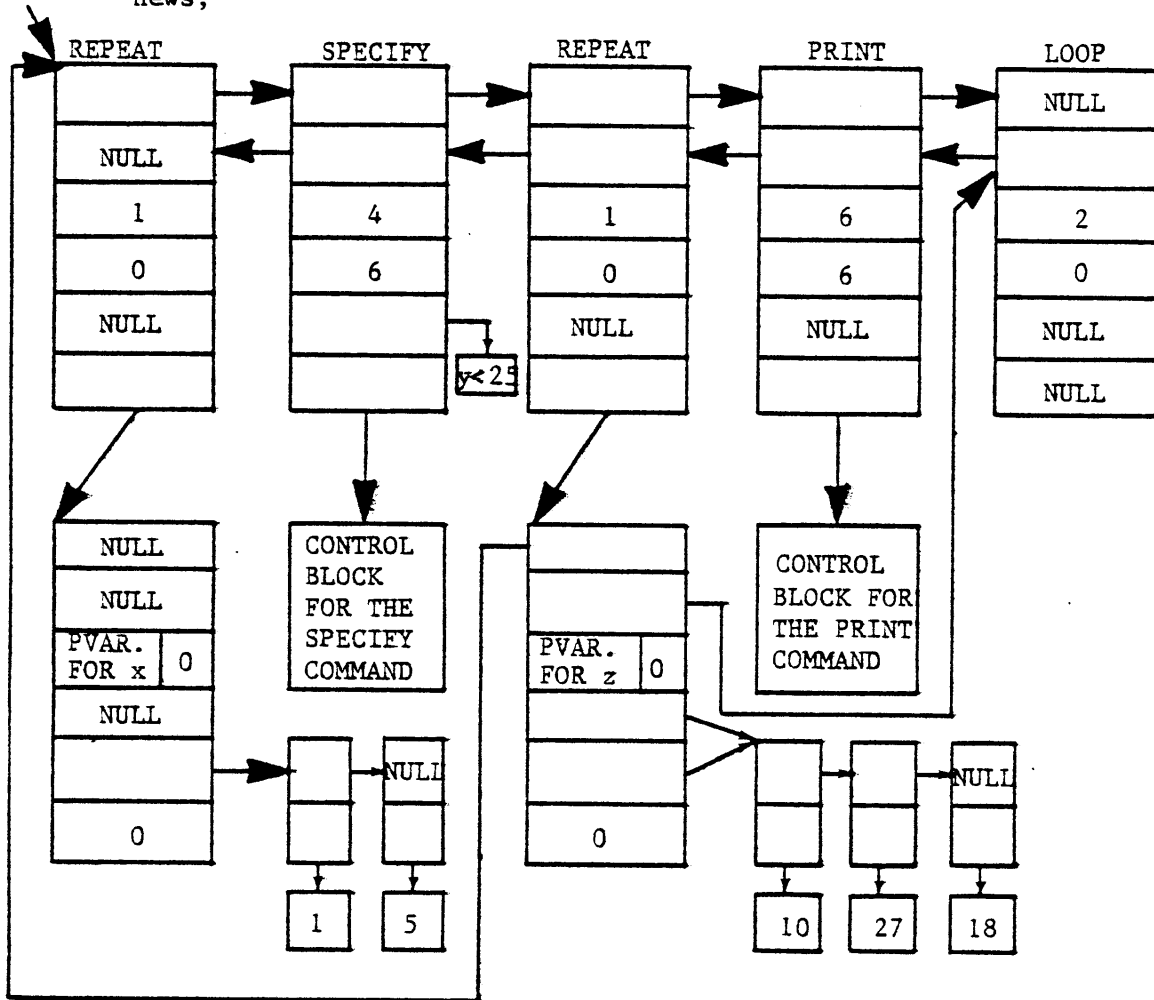
THE VARIABLE "CR" IS USED TO POINT TO THE CURRENT REPEAT COMMAND.

(cont. next page)

FIGURE 9.13 CONTINUED

EXAMPLE:

```
repeat for x from (1) to (5);
specify variable (y = sqrt(x)) if (y<25);
repeat for z = (10,27,18);
print v(x,y,z);
loop;
news;
```

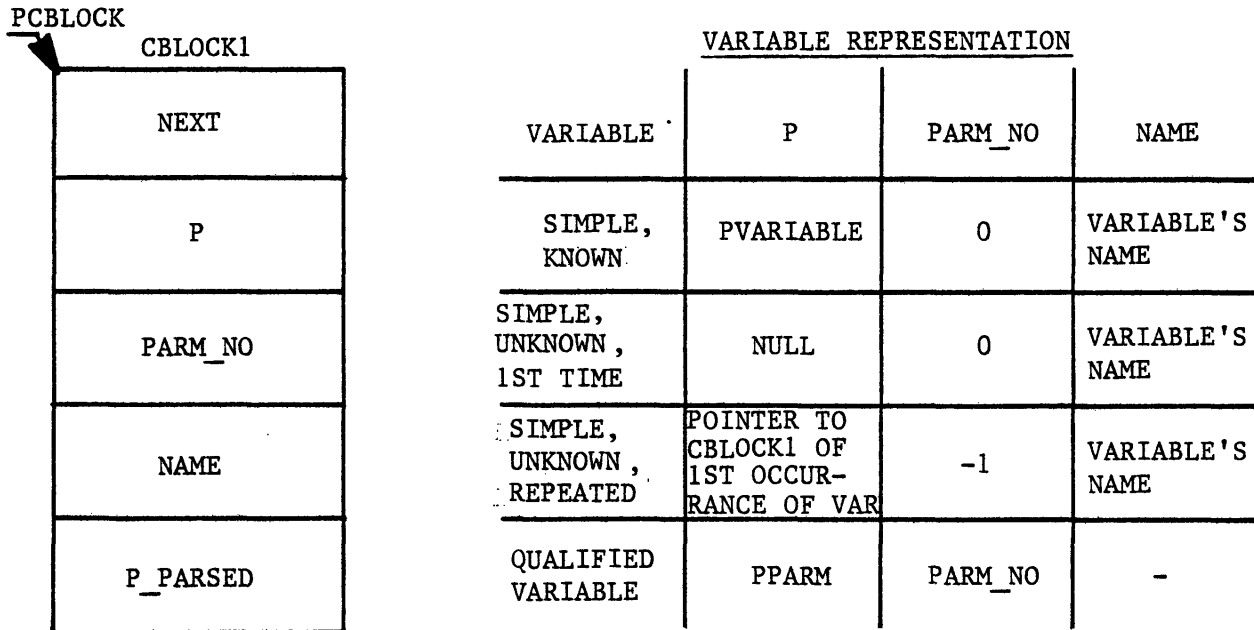


STATUS OF THE CONTROL BLOCKS AFTER THE ISSUANCE OF THE LAST COMMAND (e.g. NEWS); NOTICE THAT THE EXTERNAL REPEAT COMMAND IS NOT CLOSED YET.

FIGURE 9.14 THE CBLOCK1 STRUCTURE

USED FOR: specify, or assume variables.

SYNTAX: {specify | assume} variables (<variable> = <expression> ['dimension']
 [, <variable> = <expression> ['dimension']] *) [<if-clause>] ;



EXAMPLE: specify v(x = 4 * y, u:a.%p4 = f(x,z) - 3); \$ variable x is known \$

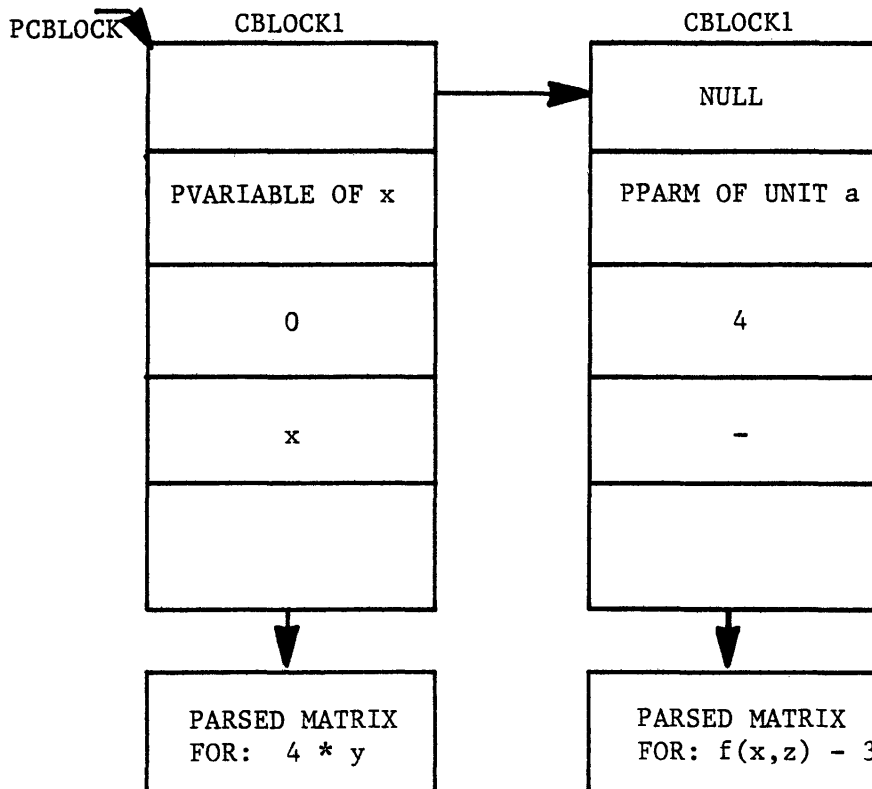
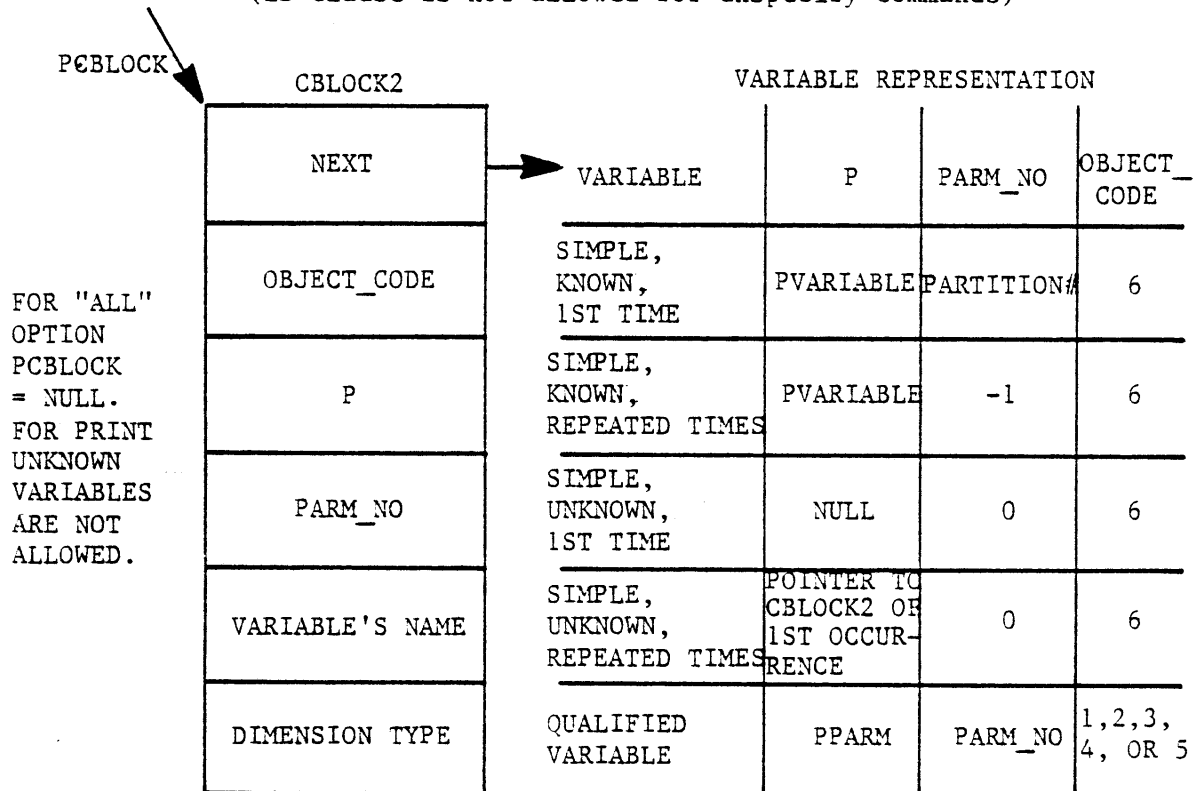


FIGURE 9.15 THE CBLOCK2 STRUCTURE

USED FOR: print, read, reada, or unspecify variables

SYNTAX: { print | read | reada | unspecify } variables
 { all | (<variable-list>) } [<if-clause>];
 (if-clause is not allowed for unspecify commands)



EXAMPLE: read v(x, u.a.t, x); \$ variable x is unknown \$

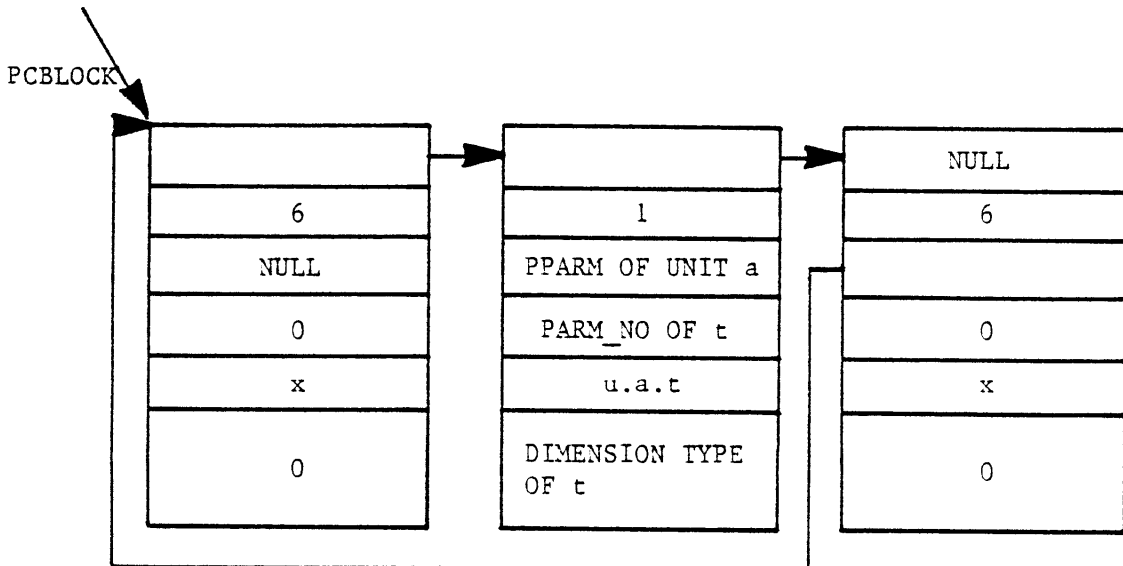


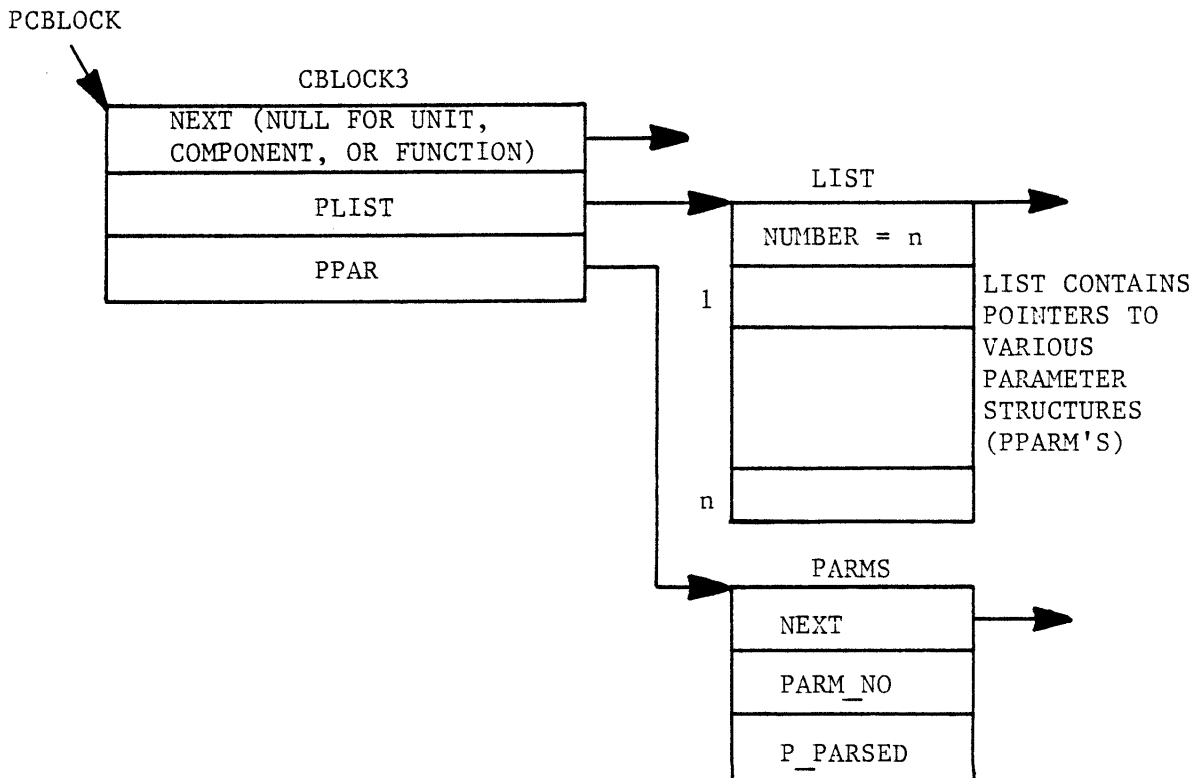
FIGURE 9.16 THE CBLOCK3 STRUCTURE

USED FOR: SPECIFY, OR ASSUME UNITS, COMPONENTS, FUNCTIONS, STREAMS OR FLOW

SYNTAX: { specify | assume } { unit | component | function } (<object-list>)
 (<parameter-specification-list>) [<if-clause>];

{ specify | assume } stream (<stream-list>) [<phase-field>]
 (<parameter-specification-list>) [[<phase-field>]
 (<parameter-specification-list>)]* [<if-clause>];

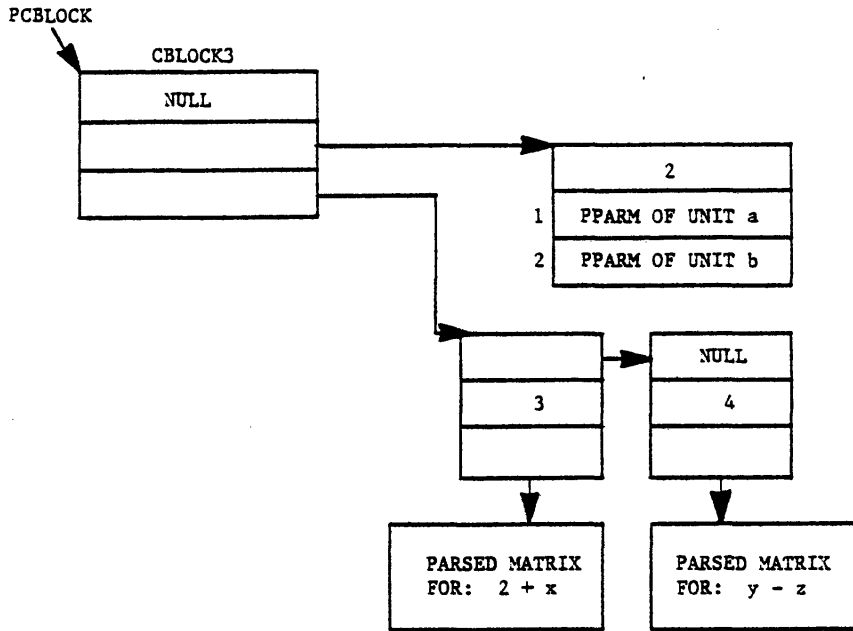
{ specify | assume } flow (<stream-list>) [<phase-field>]
 (<flow-parameter-specification-list>) [[<phase-field>]
 (<flow-parameter-specification-list>)]* [<if-clause>];



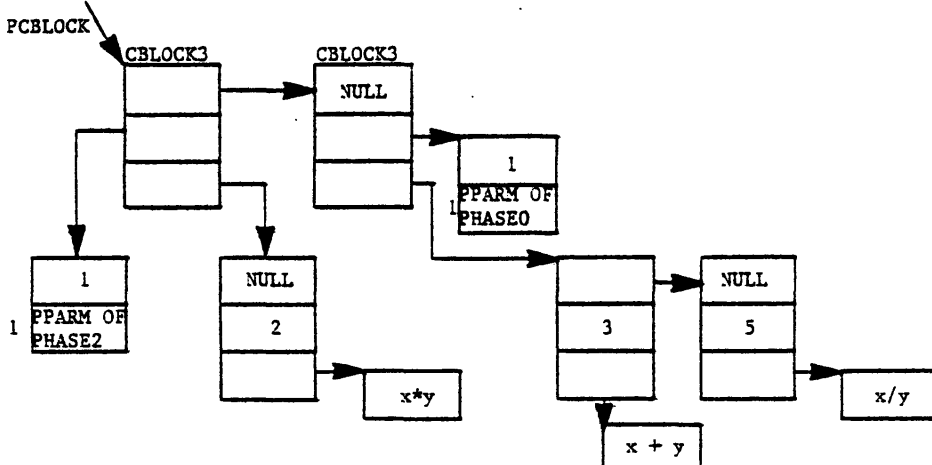
(cont. next page)

FIGURE 9.16 CONTINUED

EXAMPLE: specify units (a, b) (%p3 = 2 + x, %p4 = y - z);



EXAMPLE: specify stream (feed) phase2 (%p2 = x*y) (all = , , x + y, , x/y);



EXAMPLE: assume flow (feed) (co (%p1 = x + y, %p2 = x - y), co2 (%p1 = x*y, %p2 = x/y)); ;

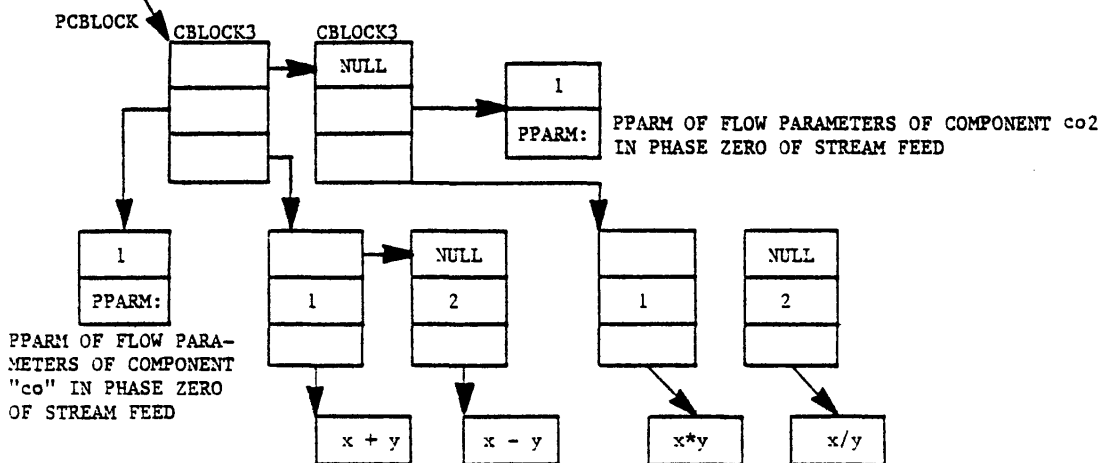
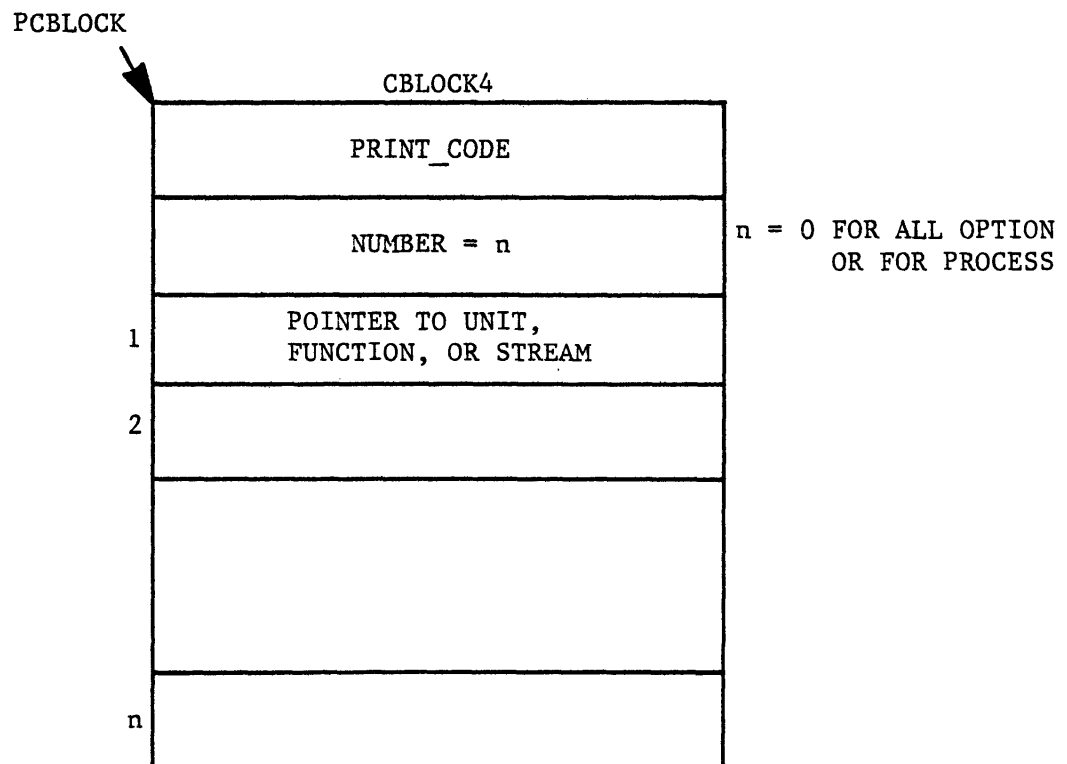


FIGURE 9.17 THE CBLOCK4 STRUCTURE

USED FOR: print units, functions, streams, flow, or process

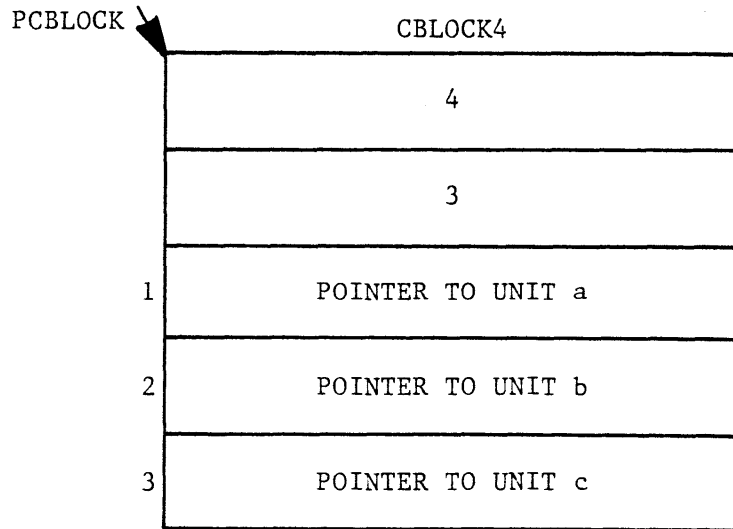
SYNTAX: print { unit | function | stream | flow } { all | (<object-list>) }
 <print-option> [<if-clause>];
 print process { flowsheet | all } [<if-clause>];



(cont. next page)

FIGURE 9.17 CONTINUED

<u>PRINT_CODE</u>	<u>PRINT_OPTION</u>
0	UNSPECIFIED OR FLOWSHEET (FOR PROCESS)
1	ASSUMED
2	SPECIFIED
3	CALCULATED
4	PARAMETERS
5	CONNECTIONS (FOR UNIT AND STREAM) STREAMS (FOR COMPONENT) FUNCTIONS (FOR FUNCTION) COMPONENTS (FOR FLOW)
6	TYPE
7	ALL



EXAMPLE:
print units (a,b,c)
parameters;

EXAMPLE: print units all all;

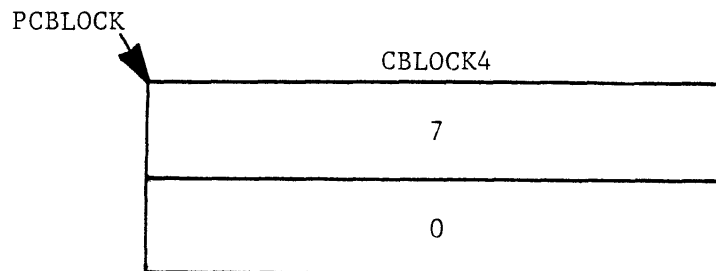
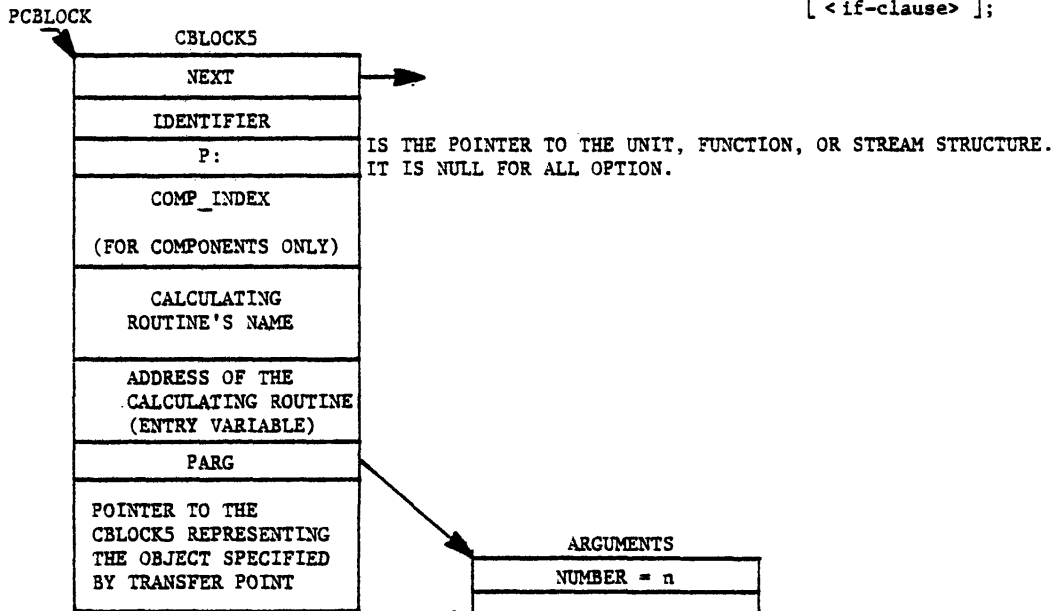


FIGURE 9.18 THE CBLOCK5 STRUCTURE

USED FOR: calculate units, components, functions, or streams.

```
SYNTAX: calculate { unit | component | function | stream } { all [ ( [ <arguments> ] ) ] |
  ( <identifier> [ ( [ <arguments> ] ) ] [ , <transfer-point> ] ) ]
  [ , <identifier> [ ( [ <arguments> ] ) ] [ , <transfer-point> ] ) ] * ) }
  [ <if-clause> ] ;
```



EXAMPLE: calculate units
(a, b(1,12,4), d(a));

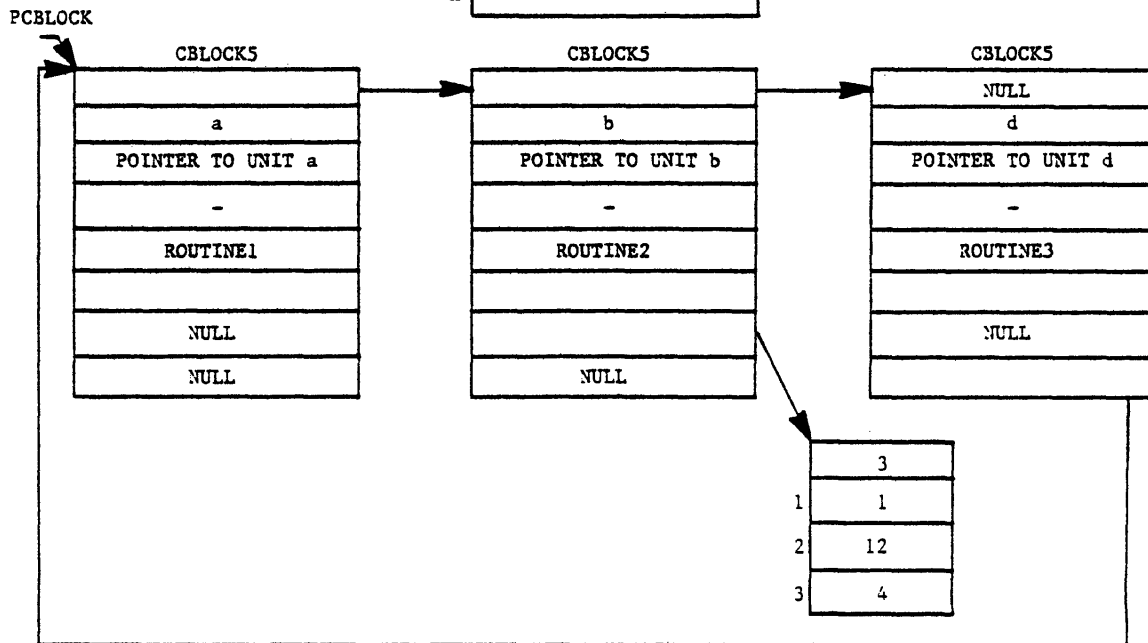
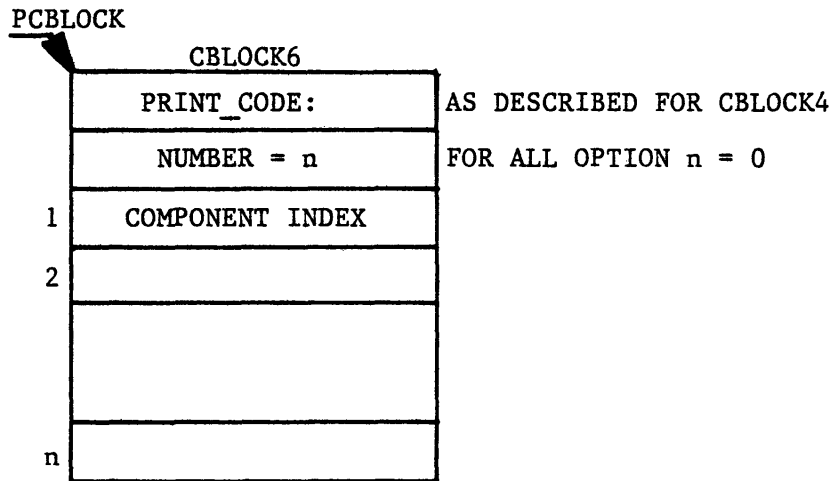


FIGURE 9.19 THE CBLOCK6 STRUCTURE

USED FOR: print components.

SYNTAX: print component { all | (<component-list>) } <print-option>
 [<if-clause>];



EXAMPLE: print components (co, co2) streams;
 \$ list the streams that contain components co or co2.\$

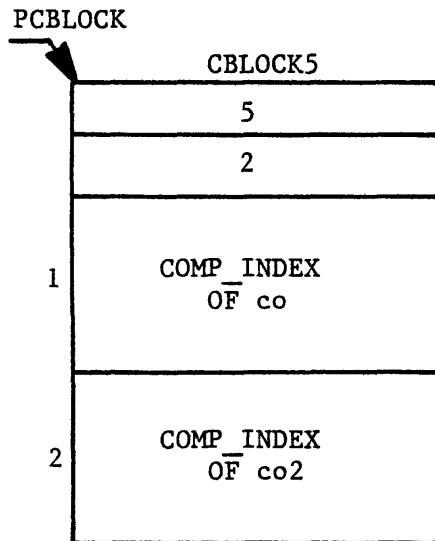


FIGURE 9.20 THE CBLOCK7 STRUCTURE

USED FOR: read, reada, or unspecify units, components, functions, streams, or flow.

SYNTAX: { read | reada | unspecify } { unit | component | function } { all | (<object-list>)
 { all | (<parameter-list>) } } [<if-clause>] ;
 { read | reada | unspecify } stream { all | (<stream-list>) [<phase-field>]
 { all | (<parameter-list>) } [[<phase-field>]
 { all | (<parameter-list>) }] * } [<if-clause>] ;
 { read | reada | unspecify } flow { all | (<stream-list>) [<phase-field>] { all |
 (<flow-parameter-list>) } [[<phase-field>]
 { all | (<flow-parameter-list>) }] * } [<if-clause>] ;

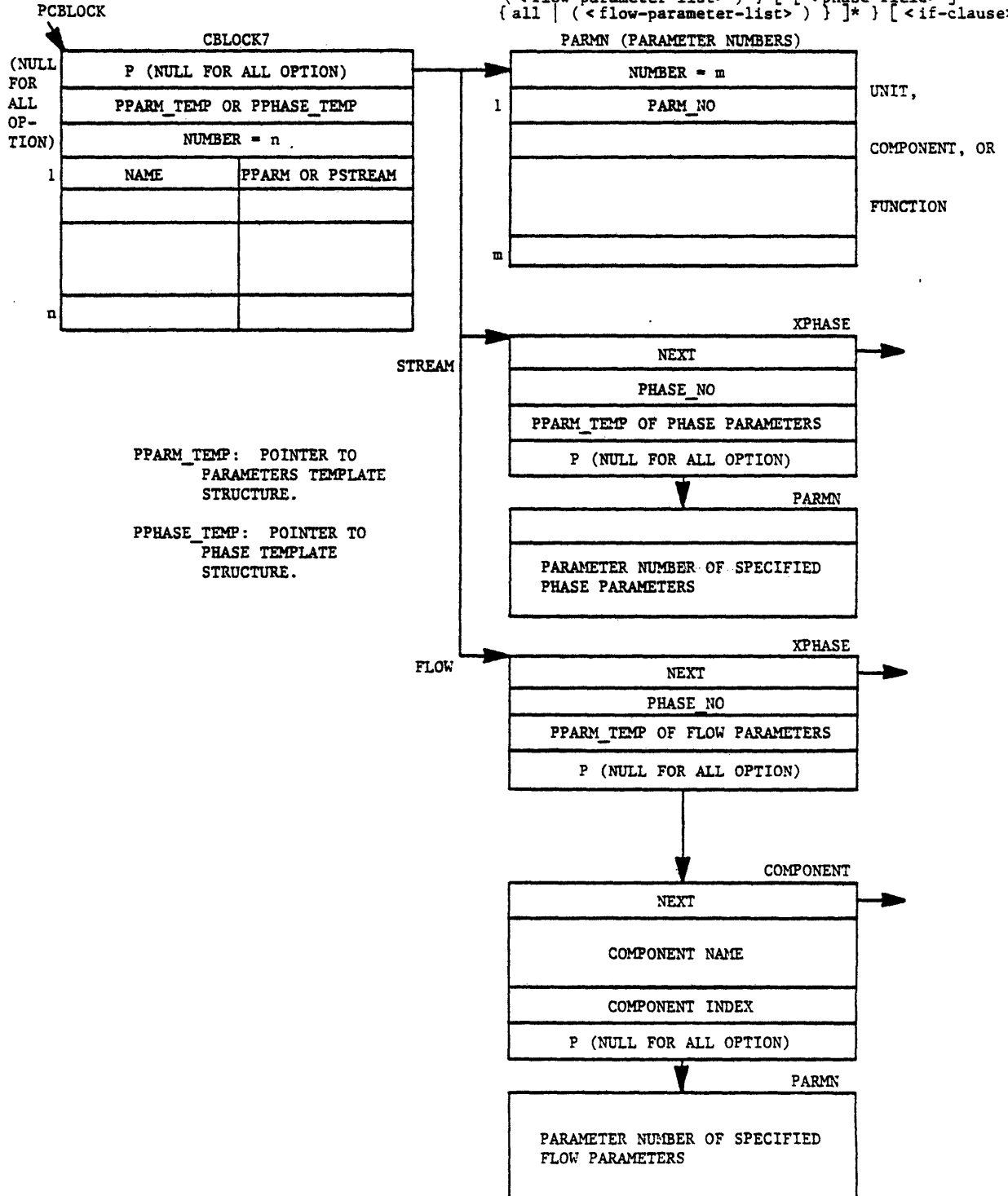
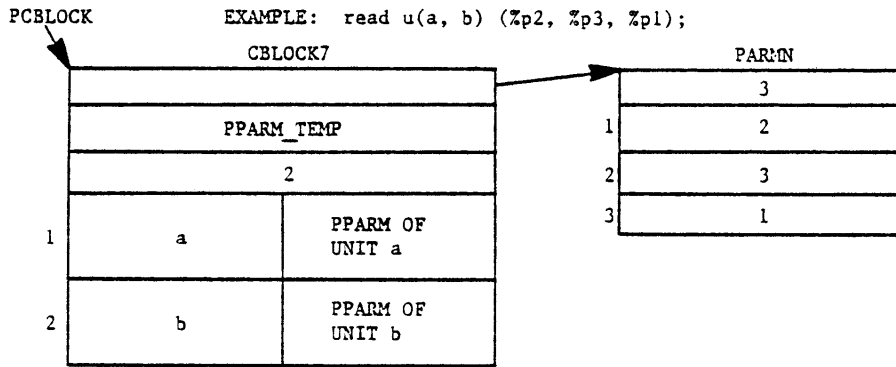
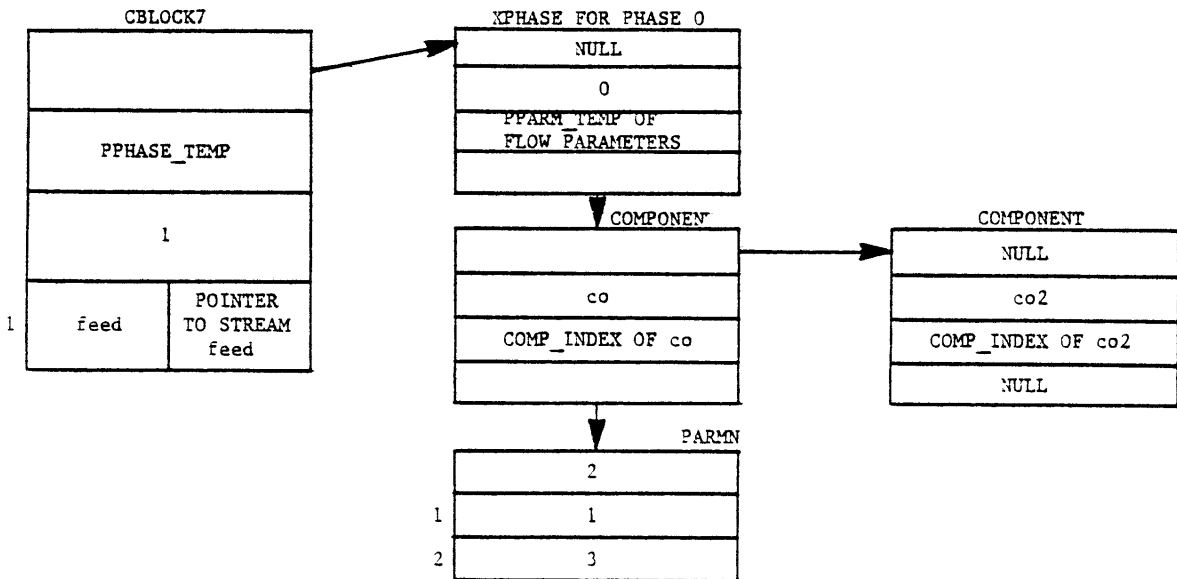
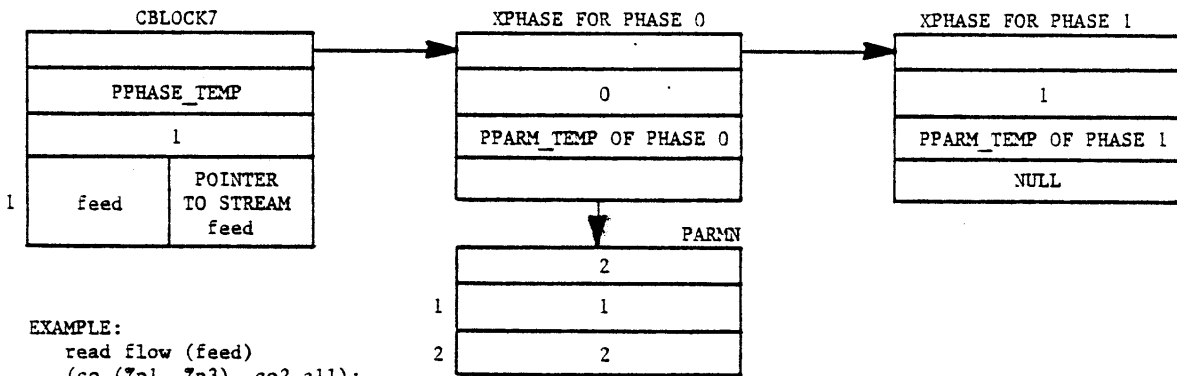


FIGURE 9.20 CONTINUED



EXAMPLE: read stream (feed) (%p1, %p2) phasel all;



9.5 Execution Phase

In this final phase the command is executed. As mentioned earlier there is a begin block associated with each command performing the syntax analysis and interpretation phase. The same begin block also monitors the execution of the command. The execution may be performed in the same block or another internal or external routine may be invoked. For commands related to component files the external routine "cdbsys" (component data base system) is called to execute the commands. The external routine "pdbsys" (process data base system) is called for command related to process files. The external routine "print_temp" is called to print information about templates in response to "printt" and "listt" commands.

The execution of commands for which intermediate forms were constructed is controlled by these intermediate forms, and monitored by some internal routines. The intermediate forms will be deleted once the execution of the command or commands within a loop is completed. The execution of other commands is performed by their associated begin blocks.

CHAPTER 10EXERCISE IN DEVELOPMENT OF TEMPLATE BASED SYSTEMS

In this chapter a general framework for the development of TBS's will be presented and the development of a number of prototype TBS's will be described.

Using GPES three prototype TBS's have been created to demonstrate the application and use of GPES and systems created by GPES. These prototype TBS's are as follows:

1. Heat Exchanger Networks Analyzer. This prototype TBS is capable of analyzing heat exchanger networks. It has been created to demonstrate that using GPES, one could develop a very simple system to solve a particular class of problems.
2. TBS-II. This prototype TBS is capable of analyzing processes with conventional liquid-vapor streams, particularly hydrocarbon processes. Currently it contains a few types of process units such as distillation columns, heat-exchangers, and isothermal flash separations, and hence it is limited to processes having only these process elements.
3. MHD. This prototype TBS is created to study MHD processes. It is an example of a case where existing simulators cannot be used to analyze processes having streams other than conventional liquid and vapor streams.

It should be noted that any of the above three TBS's could be expanded to

include additional unit operations. It is also not necessary to have different TBS's to analyze different classes of processes. In fact, a single TBS could have been created capable of analyzing all three types of processes mentioned above.

10.1 Development Process of a Template Based System

Development process of a TBS in general is shown in Figure 10.1. It consists of the following phases:

1. TBS definition (Formalism) phase.

Once the need for development of a new TBS is brought about by external factors, this would be the first phase of development. In this phase the objective of the TBS should be stated and different types of process units, streams, components, and pre-defined functions that may be required should be identified. Decisions should also be made on how to represent these elements. The mathematical models of the process units should be prepared. The standard units of measurement and other allowable options for each relevant physical dimension should also be identified. This phase not only provides input to the following two phases, but it is very useful for documentation purposes.

2. Template Definition Phase.

In this phase the TBS Administrator using the "update_tdb" program defines the templates. As described in Chapter 4 these templates are as follows:

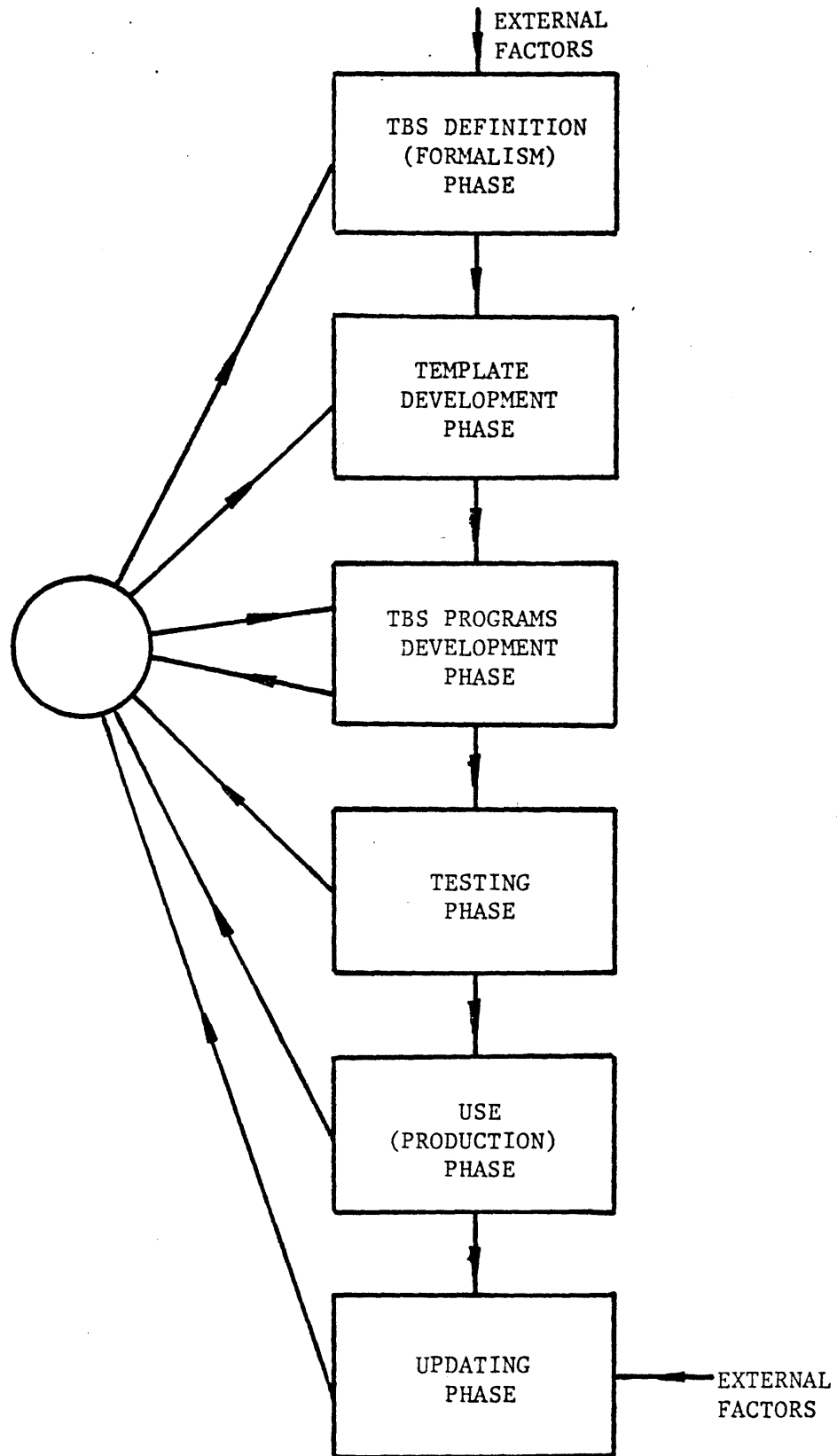


FIGURE 10.1 DEVELOPMENT PROCESS OF A TBS

- a) a template for every unit type.
- b) a template for every stream type.
- c) a template for every component type.
- d) a template for every pre-defined function type.
- e) a template for every physical dimension.
- f) a template for every physical property where the user has the option of selecting its method of estimation.
- g) a template for system control information which includes items of information such as the name of the TBS.

The TBS Administrator using any available editor should also create a text file as described in Chapter 4. This file includes items of information such as the name and address of the TBS Administrator, reported errors, news, etc.

3. TBS Programs Development Phase.

In this phase the TBS programmers develop TBS programs as described in Chapter 6. These programs mainly consist of a calculating routine for each unit type. In this effort sometimes the need for updating or modifications of efforts made in previous phases is realized, in which case the TBS Administrator has to return to one of the previous phases.

4. Testing Phase.

Before the TBS is released to the public, the TBS Administrator should test it by simulating various process flowsheets. Once

any bug is found, he may have to return to one of the previous phases for debugging.

5. Use (Production) Phase.

At this phase the TBS may be used by designers for analysis and design of various process flowsheets. In the course of the TBS usage, the users may discover some bugs in the TBS or may recognize the need for expansion or improvement of the TBS. The users should inform the TBS Administrator of the need for modification.

6. Updating Phase.

A TBS is an open-ended system which can be easily extended or modified. The need for extension or modification may be realized by inputs received from the previous phase or by other external factors. This will lead the TBS Administrator to one of the first three phases for extending or modifying the TBS.

10.2 Development of a Prototype TBS for Analyzing Heat Exchanger Networks

The development process of this simple TBS will be presented in the above framework.

1. TBS Definition Phase (formalism).

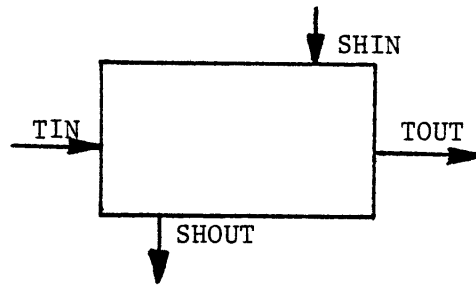
The objective of this TBS is to analyze the steady state behavior of an arbitrary heat exchanger network. Each stream in a heat exchanger network is characterized by two variables: the product of mass flow rate and specific heat (WC) and the

temperature (T). Pressure and composition are not considered to be pertinent variables.

There are four types of units: counter-current heat exchanger, mixer, divider, and convergence unit.

The counter-current heat exchanger is modeled as shown in Figure 10.2. The unit type mixer is for adding two streams and is modeled as shown in Figure 10.3. The unit type divider is for splitting one stream into two streams and is modeled as shown in Figure 10.4. Unit type convergence is for testing and promoting convergence for a recycle stream. Most chemical processes involve recycle streams. Therefore, such processes contain information recycle loops. That is, cycles for which insufficient information is available to permit equations for each unit to be solved independently. The equations for units in an information recycle loop must be solved simultaneously. One solution techniques is to "tear" one stream in the recycle loop [1,22,51,148]; that is, to guess variables of that stream. Based upon tear stream guesses, information is passed from unit to unit until new variables of the tear stream are computed.

These new values are used to repeat the calculations until convergence tolerances are satisfied. Unit type convergence is used for comparing newly computed variables (feed stream to the

Specifications:

<u>Parameters:</u>	U	overall heat-transfer coefficient (e.g., Btu/hr-ft ² -°F)
	A	area (e.g., ft ²)
<u>Connections:</u>	TIN	tube inlet
	TOUT	tube outlet
	SHIN	shell inlet
	SHOUT	shell outlet

Equations:

Material Balances:

$$WC_{TOUT} = WC_{TIN}$$

$$WC_{SHOUT} = WC_{SHIN}$$

Energy Balances:

$$T_{TOUT} = \frac{(1 - R)T_{TIN} + R(1 - F)T_{SHIN}}{1 - RF}$$

$$T_{SHOUT} = T_{SHIN} - \frac{T_{TOUT} - T_{TIN}}{R}$$

where:

$$R = \frac{WC_{SHIN}}{WC_{TIN}} ; \quad F = \exp \frac{UA}{WC_{SHIN}} (R - 1)$$

Special case (R = 1):

$$T_{TOUT} = \frac{T_{TIN} + \alpha T_{SHIN}}{1 + \alpha} ,$$

with $\alpha = UA/WC_{SHIN}$

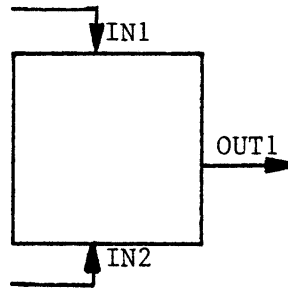
Input Variables

WC_{TIN}, T_{TIN}
 WC_{SHIN}, T_{SHIN}
 U, A

Output Variables

WC_{TOUT}, T_{TOUT}
 WC_{SHOUT}, T_{SHOUT}

Figure 10.2 Heat Exchangers TBS - Countercurrent Exchanger Model



Specifications:

Parameters: None

Connections:

IN1	first inlet stream
IN2	second inlet stream
OUT1	outlet stream

Equations:

Material Balance: $WC_{OUT1} = WC_{IN1} + WC_{IN2}$

Energy Balance:

$$T_{OUT1} = \frac{WC_{IN1} T_{IN1} + WC_{IN2} T_{IN2}}{WC_{IN1} + WC_{IN2}}$$

Input Variables

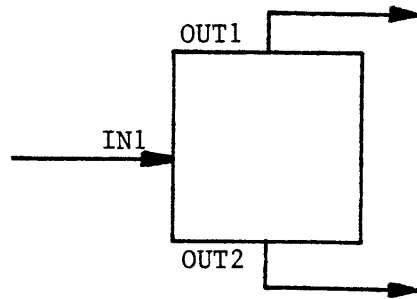
WC_{IN1}, T_{IN1}

WC_{IN2}, T_{IN2}

Output Variables

WC_{OUT1}, T_{OUT1}

Figure 10.3 Heat Exchangers TBS - Mixer Model



Specifications:

Parameters: F Fraction of inlet stream, IN1, diverted to outlet stream OUT1

Connections: IN1 inlet stream
 OUT1 first outlet stream
 OUT2 second outlet stream

Equations:

Material Balances: $WC_{OUT1} = F WC_{IN1}$
 $WC_{OUT2} = (1 - F) WC_{IN1}$

Energy Balances: $T_{OUT1} = T_{IN1}$
 $T_{OUT2} = T_{IN1}$

Input Variables

F
 WC_{IN1}, T_{IN1}

Output Variables

WC_{OUT1}, T_{OUT1}
 WC_{OUT2}, T_{OUT2}

Figure 10.4 Heat-Exchangers TBS - Divider Model.

convergence unit) with guess values (product stream from the convergence unit) and to compute new guess values when convergence tolerances (unit parameters) are not satisfied. The convergence unit is modeled as shown in Figure 10.5.

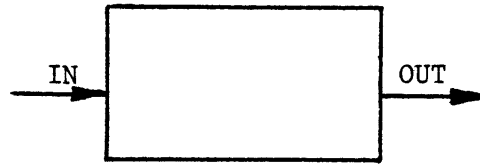
Physical dimensions pertinent to this TBS are as follows:

	<u>Standard Units</u>	<u>Optional Units</u>
1. Temperature	Degree F	Degree R, C, K
2. Area	Ft ²	
3. Heat Rate	Btu/hr	
4. Heat Transfer Coefficient	Btu/hr-ft ² -Degree F	

2. Template Definition Phase.

Using the "update_tdb" Program, the following templates have been defined:

- a. A template for the only existing stream type, std.
- b. A template for every one of the unit types: heatex, divider, mixer, and convergence.
- c. A template for every one of the four dimension types. The collection of these templates, which is referred to as a dimension table, enables the system to automatically convert the user-supplied data into standard units, if they are provided in other optional units.
- d. A template containing other miscellaneous information such as the TBS name, etc.

Specifications:Parameters:

MAXIT	Maximum number of iterations
NIT	Number of iterations
RDEV_WC	Relative deviation for stream parameter WC
ADEV_WC	Absolute deviation for stream parameter WC
RDEV_T	Relative deviation for stream parameter T
ADEV_T	Absolute deviation for stream parameter T
FLAG	Flag indicating convergence, it is positive if convergence has been achieved, it is negative otherwise.

Connections:

IN	Inlet stream
OUT	Outlet stream

Equations:

Test for convergence:

If $|WC_{IN} - WC_{OUT}| \leq (RDEV_WC)(WC_{IN}) + ADEV_WC$
 and $|T_{IN} - T_{OUT}| \leq (RDEV_T)(T_{IN}) + ADEV_T$
 then FLAG = +1, otherwise

$WC_{OUT} = WC_{IN}$, $T_{OUT} = T_{IN}$, and FLAG = -1.

Default Unit Parameters:

MAXIT = 50
 NIT = 0
 RDEV_WC = .01
 RDEV_T = .01
 ADEV_WC = .05
 ADEV_T = .05
 FLAG = -1.

Figure 10.5 Heat Exchangers TBS - Unit Convergence Model

The printout of the dimension table, the stream template and the template for unit type heatex are shown in Figure 10.6. Using the Multics editor, a text file, as described in Chapter 4, has also been created. It contains information such as the TBS Administrator name and address, news, reported errors, etc.

3. TBS Program Development Phase.

In this phase a subroutine for each process unit type is developed to represent its mathematical model. The names of these subroutines have been already supplied in the unit templates. The system will call upon these routines to solve equations for each unit. The subroutine for unit Heatex is listed in Figure 10.7.

4. Testing Phase.

In this phase the TBS has been tested by simulating various heat exchanger networks.

5. Use (Production) Phase.

Now the system is ready to be used by ultimate users of the TBS, process designers. To use the TBS they only have to know PEL (Process Engineering Language). The following example demonstrates the use of TBS and PEL.

Example

There are a number of identical heat exchangers ($A = 20 \text{ ft}^2$, $U = 10 \text{ Btu/hr-ft}^2\text{-}^\circ\text{F}$), and a number of cold streams ($WC =$

FIGURE 10.6 HEAT EXCHANGERS TBS - PRINTOUT OF SOME TEMPLATES

list units

UNIT TYPES
 heatex
 mixer
 divider
 convergence

ENTER COMMAND:
 print dimtable

DIMENSIONS TABLE

NUMBER OF DIMENSION TYPES = 4

DIMENSION TYPE	NAME	STANDARD UNITS	OPTIONS	A	B
1	temperature	f	r c k	-4.6000000e+002 3.2000000e+001 -4.6000000e+002	1.0000000e+000 1.8000000e+000 1.8000000e+000
2	area	ft2			
3	heat_rate	btu/hr			
4	heat_transfer_co	btu/hr-ft2-f			

NOTE: DIMENSION TYPE OF A DIMENSION LESS PARAMETER IS ZERO.
 ENTER COMMAND:
 print stream std

STREAM TYPE = std
 REFERENCE = a standard stream type for this TBS
 TYPE OF COMPONENTS FLOWING IN THIS STREAM = none

NUMBER OF PHASES = 0

PHASE 0 (TOTAL STREAM)

NUMBER OF PHASE PARAMETERS = 2

PARAMETER	DIMENSION TYPE
1- wc	3
2- t	1

NUMBER OF FLOW PARAMETERS = 0

PROCEDURE TO CALCULATE THE STREAM = j

FIGURE 10.6 CONTINUED
 print unit heatex

UNIT TYPE = heatex
 REFERENCE = counter_current heat exchanger.

NUMBER OF UNIT PARAMETERS = 2

PARAMETER	DIMENSION	TYPE
1- u	4	
2- a	2	

NUMBER OF INLETS = 2

INLET CONNECTIONS	STREAM TYPE
1- tin	std
2- shin	std

NUMBER OF OUTLETS = 2

OUTLET CONNECTIONS	STREAM TYPE
3- tout	std
4- shout	std

PROCEDURE TO CALCULATE THE UNIT = heatex
 MINIMUM NO OF ARGUMENTS = 0
 MAXIMUM NO OF ARGUMENTS = 1

NUMBER OF LEVELS OF CALCULATION = 1

VALUE STATUS CODES FOR LEVEL = 1

UNIT PARAMETERS	PARAMETER	CODE
	1- u	7
	2- a	7

FOR CONNECTION = tin
 THIS CONNECTION IS REQUIRED
 STREAM CONNECTED TO THIS CONNECTION = std
 PHASE 0 (TOTAL STREAM)

PHASE PARAMETER	CODE
1- wc	7
2- t	7

FOR CONNECTION = shin
 THIS CONNECTION IS REQUIRED
 STREAM CONNECTED TO THIS CONNECTION = std
 PHASE 0 (TOTAL STREAM)

PHASE PARAMETER	CODE
1- wc	7
2- t	7

FOR CONNECTION = tout
 THIS CONNECTION IS REQUIRED
 STREAM CONNECTED TO THIS CONNECTION = std
 PHASE 0 (TOTAL STREAM)

PHASE PARAMETER	CODE
1- wc	13
2- t	13

FOR CONNECTION = shout
 THIS CONNECTION IS REQUIRED
 STREAM CONNECTED TO THIS CONNECTION = std
 PHASE 0 (TOTAL STREAM)

PHASE PARAMETER	CODE
1- wc	13
2- t	13

ENTER COMMAND:

FIGURE 10.7 HEAT EXCHANGERS TBS - HEATEX CALCULATING ROUTINE

heatex.F11

05/07/78 1538.6 edt Sun

```

heatex!Proc(Funit,Parms,switch,error_switch);
  del Funit Ptr,
  Parms Ptr,
  switch bit(1),
  error_switch bit(1);
  del unit_Ptr entry(Ptr,fixed bin,Ptr,bit(1));
  del strm_Ptr entry(Ptr,fixed bin,Ptr,bit(1));
  del xput_Parm entry(Ptr,fixed bin,float bin,bit(1));
  del xset_Parm entry(Ptr,fixed bin,float bin,fixed bin,bit(1));
  del (PParm,Pstream) Ptr;
  del exp builtin;
  del vtype,
  code bit(1);
  del (u,r,ttin,ttout,tshin,tshout,wctin,wctout,wcshin,wcshout,r,f) float bin;

  /* retrieve input variables */
  call unit_Ptr(Funit,0,PParm,code);
  call xset_Parm(PParm,1,u,vtype,code);
  call xset_Parm(PParm,2,r,vtype,code);
  call unit_Ptr(Funit,1,Pstream,code);
  call strm_Ptr(Pstream,0,PParm,code);
  call xset_Parm(PParm,1,wctin,vtype,code);
  call xset_Parm(PParm,2,ttin,vtype,code);
  call unit_Ptr(Funit,2,Pstream,code);
  call strm_Ptr(Pstream,0,PParm,code);
  call xset_Parm(PParm,1,wcshin,vtype,code);
  call xset_Parm(PParm,2,tshin,vtype,code);

  /* Perform required computations */

  /* material balance */
  wctout=wctin;
  wcshout=wcshin;

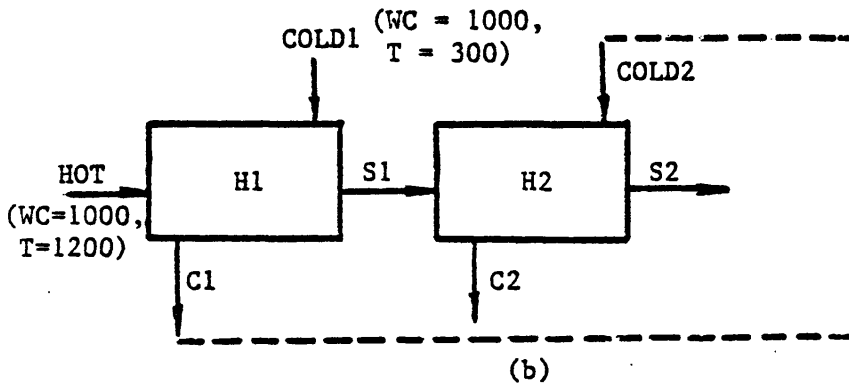
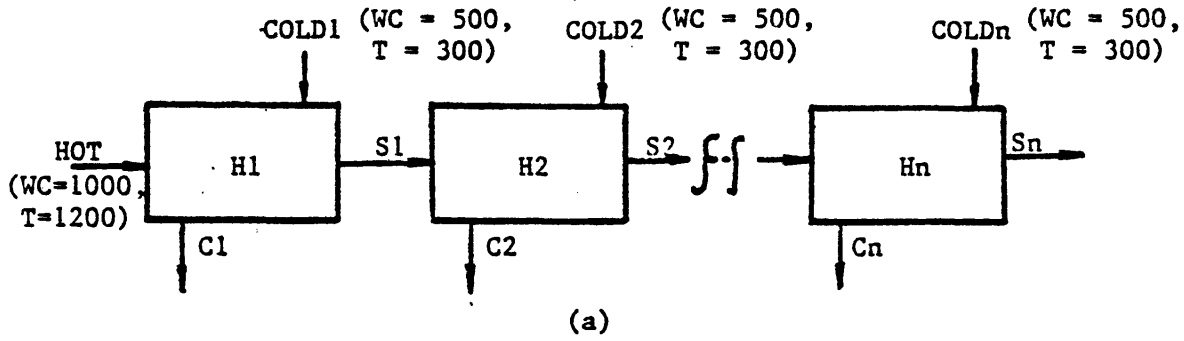
  /* energy balance */
  r=wcshin/wctin;
  f=exp(u*a/wcshin*(r-1));
  if r=1
    then ttout=(ttin+u*a/wcshin*tshin)/(1+u*a/wcshin);
    else ttout=((1-r)*ttin+r*(1-f)*tshin)/(1-r*f);
  tshout=tshin-(ttout-ttin)/r;

  /* store output variables */
  call unit_Ptr(Funit,3,Pstream,code);
  call strm_Ptr(Pstream,0,PParm,code);
  call xput_Parm(PParm,1,wctout,code);
  call xput_Parm(PParm,2,ttout,code);
  call unit_Ptr(Funit,4,Pstream,code);
  call strm_Ptr(Pstream,0,PParm,code);
  call xput_Parm(PParm,1,wcshout,code);
  call xput_Parm(PParm,2,tshout,code);

  return;
end heatex;

```

500 Btu/hr, $T = 300^{\circ}\text{F}$) to be used to cool a hot stream ($WC = 1000$ Btu/hr, $T = 1200^{\circ}\text{F}$) to below 950°F , as shown in Figure 10.8a. The problem is determining the minimum number of required heat exchangers. The computer session for solving this problem is shown in Figure 10.9. To solve this problem, one may start with one heat exchanger and increase the number of heat exchangers until the hot stream is cooled to the desired temperature. As can be seen in Figure 10.9, two heat exchangers will bring the hot stream temperature below the 950°F . Before terminating the session, the user has saved his process network, so that he may be able to continue his design effort sometime in the future. At a later session, the user wishes to investigate other design alternatives, especially those shown in Figures 10.8b and 10.8c. The PEL computer session is demonstrated in Figure 10.10. As can be seen, the output temperature for process configuration shown in 10.8b is 950°F compared to 945°F in the first configuration. To simulate the process flowsheet shown in Figure 10.8c, one observes that unit H1 and H2 cannot be calculated independently. To calculate unit H1 stream cold1 should be known. To determine stream cold1 unit H2 should be calculated which requires the stream S1 to be known. Therefore, to calculate unit H1 stream S1 should be known, but on the other



NOTES

WC IN BTU/HR
T IN °F

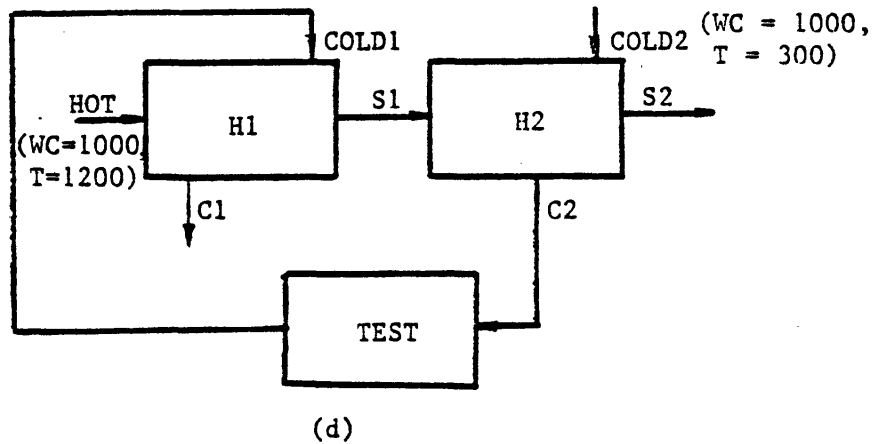
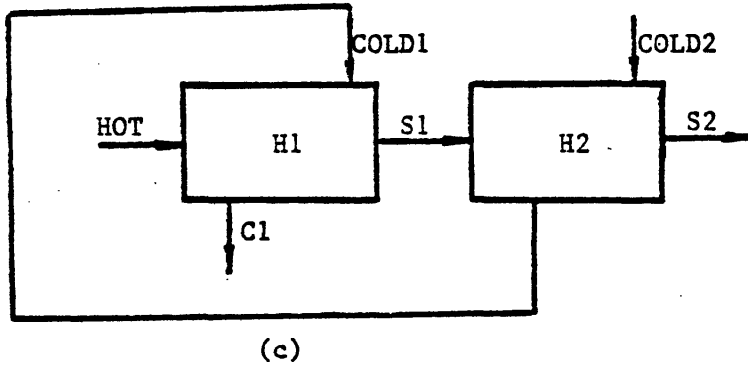


FIGURE 10.8 HEAT EXCHANGERS TBS - THE PROCESS FLOWSHEET FOR THE EXAMPLE CASE

FIGURE 10.9 HEAT EXCHANGERS TBS - A PEL COMPUTER SESSION FOR THE EXAMPLE CASE
pel brief

```

*beginning of attachment process.
**enter the name of TBS you wish to use now:heat_exchangers

*attachment process has been successful.

*new process created at:05/07/78 1831.5 edt Sun
**enter maximum number of components(0 to 200):0

**COMMAND :      $---DESCRIBE THE PROCESS CONFIGURATION---$

**continue:create unit(h1) type=heatex;

**COMMAND :create streams(hot,cold1,s1,c1);

**COMMAND :connect unit h1 at tin=hot shin=cold1 tout=s1 shout=c1;

**COMMAND :      $---SPECIFY KNOWN PARAMETERS---$

**continue:specify unit(h1) (a=20'ft2', u=10);

**COMMAND :specify stream (hot) (wc=1000'btu/hr' ,t=1200'f');

**COMMAND :specify stream(cold1)( wc=500,t=300);

**COMMAND :      $---SIMULATE THE PROCESS AND PRINT THE RESULT---$

**continue:calculate unit(h1);

*entering routine heatex          for level      1 calculation of unit      'h1'
**COMMAND :print variable(s.s1.p0.t);

          s.s1.p0.t=                      1.061892e+003 f          calculated
**COMMAND :      $---EXPAND THE PROCESS CONFIGURATION---$

**continue:let unit h2=h1;

**COMMAND :let stream cold2=cold1;

**COMMAND :connect unit h2 at all=s1,cold2,s2,c2;

*stream 's2' does not exist. a stream of type 'std' has been created.
*stream 'c2' does not exist. a stream of type 'std' has been created.
**COMMAND :      $SIMULATE AND PRINT THE RESULT---$

**continue:calc unit(h2);

*entering routine heatex          for level      1 calculation of unit      'h2'
**COMMAND :print variable(s.s2.p0.t);

          s.s2.p0.t=                      9.449777e+002 f          calculated
**COMMAND :      $---SAVE THE PROCESS---$

**continue:save process          n1at1;

***ERROR*** s 91 no process file is opened.
*command ignored.
**COMMAND :      $CREATE AND OPEN A PROCESS FILE---$

**continue:open process file(demo);

*enter the relative or absolute pathname of the directory containing the process file 'demo'
(if it is the same as your working directory ,>udd>ICPES>Arab-Ismaili, enter a null line):

*process file 'demo' does not exist.
**if you wish a new one be created enter yes ,otherwise no:yes

*process file 'demo' is opened.
**COMMAND :save process n1at1;

*process has been saved.
**COMMAND :end;

*thank you Arab-Ismaili for trying GPES come back soon, bye!
r 1838 4.330 153.635 2482

```

FIGURE 10.10 HEAT EXCHANGERS TBS - ANOTHER PEL COMPUTER SESSION FOR THE EXAMPLE CASE
pel brief

```

*beginning of attachment process.
**enter the name of TBS you wish to use now:heat_exchangers

*attachment process has been successful.

*new process created at:05/07/78 1844.1 edt Sun
**enter maximum number of components(0 to 200):0

**COMMAND :      $---LOAD THE PROCESS---$

**continue:open process file(demo);

**enter the relative or absolute pathname of the directory containing the process file 'demo'
(if it is the same as your working directory ,>udd>ICPES>Arab-Ismaili, enter a null line):

*process file 'demo' is opened.
**COMMAND :load process net1;

*process 'net1' has been created at:05/07/78 1831.5 edt Sun
  by system: GPES          serial_no: 1 compat_level: 1
  and TBS: heat_exchangers serial_no: 1 compat_level: 1
  and it has not been accessed by any other incompatible system since.
*process has been loaded.
**COMMAND :      $---SIMULATE THE PROCESS FOR CONFIGURATION SHOWN IN FIGURE 10.8 b---$

**continue:specify stream(cold1)(wc=1000);

**COMMAND :calc u(h1);

*entering routine heatex          for level 1 calculation of unit 'h1'
**COMMAND :let stream cold2=c1;

**COMMAND :calc u(h2);

*entering routine heatex          for level 1 calculation of unit 'h2'
**COMMAND :print v(s.s2.p0.t);

      s.s2.p0.t=          9.500000e+002 f          calculated
**COMMAND :      $SIMULATE THE PROCESS FOR CONFIGURATION SHOWN IN FIGURE 10.8 d---$

**continue:create unit(test) type=convergence;

**COMMAND :connect unit test at in=c2 out=cold1;

**COMMAND :      $---use the default values for parameters of unit test---$

**continue:calc unit(test(2));

*entering routine convs          for level 2 calculation of unit 'test'
**COMMAND :specify stream (cold2)(wc=1000,t=300);

**COMMAND :      $---assume the initial values of tear stream---$

**continue:assume stream(cold1)(wc=1000,t=400);

**COMMAND :calculate units(h1,h2,test(,h1));

*entering routine heatex          for level 1 calculation of unit 'h1'
*entering routine heatex          for level 1 calculation of unit 'h2'
*entering routine convs          for level 1 calculation of unit 'test'
*entering routine heatex          for level 1 calculation of unit 'h1'
*entering routine heatex          for level 1 calculation of unit 'h2'
*entering routine convs          for level 1 calculation of unit 'test'
**COMMAND :      $AS CAN BE SEEN CONVERGENCE HAS BEEN ACHIEVED IN TWO ITERATIONS---$

**continue:print variable(s.s2.p0.t);

      s.s2.p0.t=          9.427469e+002 f          calculated
**COMMAND :      $---SAVE THE NEW PROCESS FOR FUTURE STUDIES---$

**continue:save process net1 override;

*process has been saved.
**COMMAND :end;

*thank you Arab-Ismaili for trying GPES come back soon, bye!
r 1853 1.586 90.254 1479

```

hand, to find the stream S1 requires the calculation of unit H1. Hence, it can be seen that the process flowsheet contains an information recycle loop; that is, too few stream variables are known to permit equations for each unit to be solved independently. One solution technique is to tear one stream in the recycle loop; that is, to guess variables of that stream. Based upon tear stream guesses, information is passed from unit to unit until new values of the variables of the tear stream are computed. These new values are used to repeat the calculations until convergence tolerances are satisfied. This has been done as shown in Figure 10.8d and demonstrated in Figure 10.10.

As can be seen, convergence has been achieved in two iterations. The output temperature is 943^oF which is lower than the two previous outcomes.

6. Updating Phase (TBS Administrator's Task)

A TBS is an open ended system which can easily be extended or modified. Suppose a new unit type is to be added to the TBS. The unit type is adjuster, which heats or cools a stream to a specified temperature. The unit model is shown in Figure 10.11. A new template defining this new unit type has been added to the template based system as shown in Figure 10.12 and a subroutine representing the mathematical model of the unit

Specifications:

Parameters: Q Heat added (removed) (e.g., Btu/hr)

Connections: IN Inlet stream
 OUT Outlet stream

Equations:

Material Balance: $WC_{OUT} = WC_{IN}$

Energy Balance: $Q = (T_{OUT} - T_{IN})WC_{IN}$

Input Variables

WC_{IN}

T_{IN}

T_{OUT}

Output Variables

WC_{OUT}

Q

Figure 10.11 Heat Exchangers TBS - Adjuster Model

FIGURE 10.12 HEAT EXCHANGERS TBS - INSERTING A TEMPLATE

```

update_tdb

IS IT A NEW TDB?  yes OR no?no

ENTER THE ABSOLUTE  PATH NAME OF THE DIRECTORY CONTAINING THE DATABASE SEGMENTS:
>udd>ICPES>Arab-Ismaili>heat_exchangers

ENTER COMMAND:
insert unit adjuster

ENTER REFERENCE:
Heats or cools a stream to a specified temperature.

ENTER NUMBER OF UNIT PARAMETERS:
1

ENTER PARM NAMES AND DIM TYPES:
      NAME      DIM TYPE
1:a           3

ENTER NUMBER OF CONNECTIONS:
2

ENTER NUMBER OF INLETS:
1

ENTER CON NAMES AND STREAM TYPES:
      NAME      STRM TYPE
1:in          std
2:out          std

ENTER PROCEDURE NAME TO CALCULATE THE UNIT:
adjuster

ENTER MIN NUMBER OF ARGUMENTS:
0

ENTER MAX NUMBER OF ARGUMENTS:
1

ENTER NUMBER OF LEVELS OF CALCULATION:
1

VALUE STATUS CODES FOR LEVEL      1
ENTER VALUE STATUS CODE FOR      1 UNIT PARAMETERS
1:13

FOR CONNECTION in
ENTER CONNECTION STATUS:1
FOR PHASE      0
ENTER VALUE STATUS CODE FOR      2 PHASE PARAMETERS
1:7
2:7

FOR CONNECTION out
ENTER CONNECTION STATUS:1
FOR PHASE      0
ENTER VALUE STATUS CODE FOR      2 PHASE PARAMETERS
1:13
2:7

OK?  yes OR no?yes

ENTER COMMAND:
end

-----DATABASE INCONSISTENCIES-----

-----*****----- NO DATABASE INCONSISTENCIES -----*****-----

DO YOU WISH TO EXIT?  yes OR no ?
yes

ENTER DEFAULT UNIT TYPE:
heatex

```

has been developed. The users of the TBS may now incorporate units of this type in their process flowsheets when such a need arises.

10.3 A Prototype TBS for Hydrocarbon Processes

The objective of this TBS is to analyze the steady state behavior of hydrocarbon processes. One stream type, one component type, and a number of unit types will be required for such a TBS. It should be clear that a process flowsheet may contain any number of streams, components or units of the specified types.

The stream type which is referred to as "std" (standard) stream is a conventional vapor-liquid stream. This stream in addition to phase zero which represents the total stream, has two more phases to represent the vapor and liquid phases. Phase zero has five phase parameters and one flow parameter. The phase parameters are:

- 1-t Temperature in degrees Kelvin
- 2-p Pressure in atmospheres
- 3-f Total molal flow rate in Kgmol/hr
- 5-h Molal enthalpy in Kcal/Kgmol
- 5-vf Vapor fraction.

The flow of components in the total stream is represented by their

FIGURE 10.13 TBS-II - TEMPLATES FOR CONTROL INFORMATION, DIMENSION TABLE AND PROPERTY ESTIMATION METHODS TABLE

SYSTEM CONTROL INFORMATION

SYSTEM NAME = TBS-II
 SERIAL NUMBER = 1
 COMPATIBILITY LEVEL = 1

PROCEDURE TO CALCULATE ALL UNITS = ;
 PROCEDURE TO CALCULATE ALL STREAMS = ;
 PROCEDURE TO CALCULATE ALL COMPONENTS = ;
 PROCEDURE TO CALCULATE ALL FUNCTIONS = ;

DEFAULT NUMBER OF SIGNIFICANT DIGITS = 6
 DEFAULT NUMBER OF DECIMAL DIGITS = 5
 DEFAULT DEBUGGING FLAG = 0

UNIT TYPES

iflash
 heatex
 splitter
 convs
 distillation
 add

STREAM TYPES

std

COMPONENT TYPES

one

FUNCTION TYPES

11

DIMENSIONS TABLE

NUMBER OF DIMENSION TYPES = 7

DIMENSION TYPE	NAME	STANDARD UNITS	OPTIONS	A	B
1	temperature	k	c	2.7300000e+002	1.0000000e+000
			r	0.0000000e+000	5.5555556e-001
			f	2.5555556e+002	5.5555556e-001
2	pressure	ata	psia	0.0000000e+000	6.8050000e-002
			atms	0.0000000e+000	1.3157900e-003
3	flow_rate	ksmol/hr	ibsmole/hr	0.0000000e+000	4.5359000e-001
4	enthalpy_rate	kcal/hr			
5	area	m2	ft2	0.0000000e+000	9.2903040e-002
6	heatex_coeff	kcal/hr/m2/k			
7	enthalpy/mol	kcal/ksmol			

NOTE: DIMENSION TYPE OF A DIMENSION LESS PARAMETER IS ZERO.

PROPERTY ESTIMATION METHODS TABLE

NUMBER OF PROPERTIES = 1

PROPERTY NUMBER	PROPERTY	NO. OF OPTIONS	DEFAULT METHOD
1	pvap	2	1

mole fractions. Thus, the only flow parameter of phase zero is the mole fraction (z).

Phase one represents the vapor phase and has zero phase parameters and one flow parameter. The latter is the mole fraction in the vapor phase.

Phase two represents the liquid phase and similarly has zero phase parameters and one flow parameter. The latter is the mole fraction in liquid phase (x).

A calculating routine is associated with this stream type which has two levels of calculation. Level one of calculation normalizes the total stream mole fractions, and level two performs an equilibrium calculation. Given the stream temperature (t), pressure (p), and total stream mole fractions (z's), level two of calculation determines the molal enthalpy (h), vapor fraction (vf), and compositions in each phase (y's and x's). The template for the stream type is shown in Figure 10.14.

The component type "one" which is the only type component in this TBS is represented by the following fifteen parameters:

1 - molwt	molecular weight
2 - tb	normal boiling point, K
3 - tc	critical temperature, K
4 - pc	critical pressure, K
5 - zc	critical compressibility
6 - omega	Pitzer's acentric factor

TABLE 10.14 TBS-II - STREAM TEMPLATE

STREAM TYPE = std
 REFERENCE = Standard Stream- Vapor-Liquid Stream
 TYPE OF COMPONENTS FLOWING IN THIS STREAM = one

NUMBER OF PHASES = 2

PHASE 0 (TOTAL STREAM)

NUMBER OF PHASE PARAMETERS = 5

PARAMETER	DIMENSION	TYPE
1- t	1	
2- P	2	
3- f	3	
4- h	7	
5- vf	0	

NUMBER OF FLOW PARAMETERS = 1

PARAMETER	DIMENSION	TYPE
1- z	0	

PHASE 1

NUMBER OF PHASE PARAMETERS = 0

NUMBER OF FLOW PARAMETERS = 1

PARAMETER	DIMENSION	TYPE
1- v	0	

PHASE 2

NUMBER OF PHASE PARAMETERS = 0

NUMBER OF FLOW PARAMETERS = 1

PARAMETER	DIMENSION	TYPE
1- x	0	

PROCEDURE TO CALCULATE THE STREAM = strncalc

MINIMUM NO OF ARGUMENTS = 0

MAXIMUM NO OF ARGUMENTS = 1

NUMBER OF LEVELS OF CALCULATION = 2

VALUE STATUS CODES FOR LEVEL = 1

VALUE STATUS CODE FOR ALL PARAMETERS OF ALL COMPONENTS
 FLOWING IN THIS STREAM = 15

PHASE 0 (TOTAL STREAM)

PHASE PARAMETER	CODE
1- t	15
2- P	15
3- f	15
4- h	15
5- vf	15
FLOW PARAMETER	CODE
1- z	5

PHASE 1

FLOW PARAMETER	CODE
1- v	15

PHASE 2

FLOW PARAMETER	CODE
1- x	15

VALUE STATUS CODES FOR LEVEL = 2

VALUE STATUS CODE FOR ALL PARAMETERS OF ALL COMPONENTS
 FLOWING IN THIS STREAM = 7

PHASE 0 (TOTAL STREAM)

PHASE PARAMETER	CODE
1- t	7
2- P	7
3- f	15
4- h	13
5- vf	13
FLOW PARAMETER	CODE
1- z	7

PHASE 1

FLOW PARAMETER	CODE
1- v	13

PHASE 2

FLOW PARAMETER	CODE
1- x	13

- 7 - cpvapa constants in ideal gas heat capacity equation
- 8 - cpvapb with cp in kilocalories per kg-mole Kelvin and
- 9 - cpvapc and t in Kelvins:
- 10 - cpvapd $cp = cpvapa + (cpvapb)t + (cpvapc)t^2 + (cpvapd)t^3$
- 11 - delhg standard enthalpy of formation at 298°K,
Kcal/Kg-mole
- 12 - anta Antoine's vapor pressure equation coefficients, with
- 13 - antb vapor pressure in millimeters of mercury and t
- 14 - antc in Kelvins:
 $\ln(\text{vapor pressure}) = anta - antb/(t + antc)$
- 15 - hv Heat of vaporization at normal boiling point,
kcal/kg-mole

The choice of the above parameters to represent the component has been based on the availability of data and requirements of thermodynamic property estimation routines. The latter assume ideal thermodynamic behavior (i.e. ideal gas and ideal solution) for both liquids and vapors. The template for the component type is shown in Figure 10.15.

Although currently there are six unit types for this TBS, more unit types can be easily added. The unit types are as follows:

- 1) Isothermal flash (iflash). It determines the quantity and composition of liquid and vapor streams resulting when a feed stream is flashed at a specified temperature and

FIGURE 10.15 TBS-II - COMPONENT AND FUNCTION TEMPLATES

COMPONENT TYPE = one
 REFERENCE = Component of Type one

NUMBER OF COMPONENT PARAMETERS = 15

	NAME	DIM TYPE
1-	molwt	0
2-	tb	1
3-	tc	1
4-	pc	2
5-	zc	0
6-	omega	0
7-	cpvapa	0
8-	cpvapb	0
9-	cpvapc	0
10-	cpvapd	0
11-	delhs	7
12-	anta	0
13-	antb	0
14-	antc	0
15-	hv	7

PROCEDURE TO CALCULATE THE COMPONENT = ;

ENTER COMMAND:
 print function 11

FUNCTION TYPE = 11
 REFERENCE = Least Squares Line Fitting

NUMBER OF FUNCTION PARAMETERS = 2

	PARAMETER	DIMENSION TYPE
1-	a0	0
2-	a1	0

PROCEDURE TO EVALUATE THE FUNCTION = 11eval
 NUMBER OF ARGUMENTS = 1

PROCEDURE TO CALCULATE THE FUNCTION = 11calc
 MINIMUM NO OF ARGUMENTS = 0
 MAXIMUM NO OF ARGUMENTS = 1

NUMBER OF LEVELS OF CALCULATION = 1

	PARAMETER	CODE
1-	a0	13
2-	a1	13

pressure. The template for this unit type is shown in Figure 10.16. The calculating routine for this unit type and the calculating routine for the stream type are both entries of a single program which is listed in Figure 10.17.

- 2) Distillation. The rigorous multicomponent distillation method of Thiele-Geddes was used in the calculating routine of this unit. The detailed equations and algorithms are given in Perry [127] (equations 13-115 to 13-134). The convergence forcing procedure, the theta-method, was also used.

The routine is rigorous in the sense that heat balances, material balances, and equilibrium relations are applied at each stage.

The following restrictions apply to this unit type:

- a) single saturated liquid feed
- b) total condenser
- c) partial reboiler
- d) top and bottom drawoffs only.

The template for this unit type is shown in Figure 10.18.

- 3) Heat Exchanger (heatex). It determines the stream outlet temperatures and thermal duty for a counter-current one shell and one tube heat exchanger involving single phase fluids undergoing no phase change.

FIGURE 10.6 TBS-II - ISOTHERMAL FLASH UNIT TEMPLATE

UNIT TYPE = iflash
 REFERENCE = Isothermal flash calculation

NUMBER OF UNIT PARAMETERS = 4

PARAMETER	DIMENSION	TYPE
1- temp	1	
2- pres	2	
3- max_liter	0	
4- tolerance	0	

NUMBER OF INLETS = 1

INLET CONNECTIONS	STREAM TYPE
1- in	std

NUMBER OF OUTLETS = 2

OUTLET CONNECTIONS	STREAM TYPE
2- overhead	std
3- bottoms	std

PROCEDURE TO CALCULATE THE UNIT = iflash

MINIMUM NO OF ARGUMENTS = 0

MAXIMUM NO OF ARGUMENTS = 1

NUMBER OF LEVELS OF CALCULATION = 1

VALUE STATUS CODES FOR LEVEL = 1

UNIT PARAMETERS	PARAMETER	CODE
1- temp	7	
2- pres	7	
3- max_liter	7	
4- tolerance	7	

FOR CONNECTION = in

THIS CONNECTION IS REQUIRED

STREAM CONNECTED TO THIS CONNECTION = std

VALUE STATUS CODE OF ALL PARAMETERS OF ALL COMPONENTS

FLOWING IN THIS STREAM = 7

PHASE 0 (TOTAL STREAM)

PHASE PARAMETER	CODE
1- t	15
2- p	15
3- f	7
4- h	15
5- vf	15
FLOW PARAMETER	CODE
1- z	7

PHASE 1

FLOW PARAMETER	CODE
1- y	15

PHASE 2

FLOW PARAMETER	CODE
1- x	15

FOR CONNECTION = overhead

THIS CONNECTION IS REQUIRED

STREAM CONNECTED TO THIS CONNECTION = std

VALUE STATUS CODE OF ALL PARAMETERS OF ALL COMPONENTS

FLOWING IN THIS STREAM = 7

PHASE 0 (TOTAL STREAM)

PHASE PARAMETER	CODE
1- t	13
2- p	13
3- f	13
4- h	13
5- vf	13
FLOW PARAMETER	CODE
1- z	13

PHASE 1

FLOW PARAMETER	CODE
1- y	13

PHASE 2

FLOW PARAMETER	CODE
1- x	13

FOR CONNECTION = bottoms

THIS CONNECTION IS REQUIRED

STREAM CONNECTED TO THIS CONNECTION = std

VALUE STATUS CODE OF ALL PARAMETERS OF ALL COMPONENTS

FLOWING IN THIS STREAM = 7

PHASE 0 (TOTAL STREAM)

PHASE PARAMETER	CODE
1- t	15
2- p	15
3- f	13
4- h	13
5- vf	13
FLOW PARAMETER	CODE
1- z	13

PHASE 1

FLOW PARAMETER	CODE
1- y	13

PHASE 2

FLOW PARAMETER	CODE
----------------	------

FIGURE 10.17 CALCULATING ROUTINES FOR STREAM AND ISOTHERMAL FALSH UNIT

```

strmcalc:proc(pstream,parms,switch,error_switch);

    /*DECLARATIONS*/

    dcl (pstream,parms)ptr;
    dcl (switch,error_switch) bit(1);
    dcl code bit(1);

    dcl (level ,i,vtype);
    dcl (sum,t,r,h,f,vf) float bin(63);
    dcl max_nc ext;
    dcl (z(max_nc),x(max_nc),y(max_nc)) float bin(63);
    dcl vtv(max_nc);
    dcl pparm ptr;

    dcl strm_ptr entry(ptr,fixed bin,ptr,bit(1));
    dcl set_ars entry(ptr ,fixed bin,fixed bin,bit(1));
    dcl set_parm entry(ptr,fixed bin,float bin(63),fixed bin,bit(1));
    dcl put_parm entry(ptr ,fixed bin,float bin(63),bit(1));
    dcl set_fparmacs entry(ptr,fixed bin,fixed bin,dim(*) float bin(63),
        dim(*) fixed bin,bit(1));
    dcl put_fparmacs entry(ptr,fixed bin,fixed bin,dim(*) float bin(63),bit(1));
    dcl equilb entry(dim(*) float bin(63),dim(*) fixed bin,float bin(63),float bin(63),
        dim(*) float bin(63),dim(*) float bin(63),float bin(63),float bin(63),
        fixed bin,bit(1));
    dcl enthal entry(dim(*) float bin(63),dim(*) fixed bin,float bin(63),float bin(63),bit(1),
        bit(1))
        returns(float bin(63));
    dcl svsprint output file stream;

    /*STREAM CALCULATING ROUTINE*/

    /* find the level of calculation */

    call set_ars(parms,1,level,code);
    goto 1(level);

1(1):    /*LEVEL 1: normalize mole fractions */

        /* retrieve input variables */
    call set_fparmacs(pstream,0,1,z,vtv,code);

        /* perform the calculations */
    sum=0;
    do i=1 to max_nc;
    if vtv(i)=-1
        then if z(i)<0
            then do;
                put skip edit('STRM CALC: ONE OR MORE OF MOLE FRACTIONS ARE NEGATIVE ')(a);
                error_switch='1'b;
                return;
            end;
            else sum=sum+z(i);
    end;

        /* store the results */

    call put_fparmacs(pstream,0,1,z/sum,code);
    return;

```

FIGURE 10.17 CONTINUED

```

1(2): /*LEVEL 2: FIND ENTHALPY AND EQUILIBRIUM COMPOSITIONS */

      /* set input variables */
call strn_ptr(pstream,0,pparm,code);
call set_parm(pparm,1,t,vtype,code);
call set_parm(pparm,2,p,vtype,code);
call set_fparamacs(pstream,0,1,z,vtv,code);

      /* perform the computations */

call ealib(z,vtv,t,p,x,y,vf,.01,50,code);
if code
  then do;
    error_switch='1'b;
    put skip edit('STRM CALC: NO CONVERGENCE')(a);
    return;
  end;
h=vf*enthal(y,vtv,t,p,'1'b,code)+ (1-vf)*enthal(x,vtv,t,p,'0'b,code);

      /* store the output variables */
call put_parm(pparm,4,h,code);
call put_parm(pparm,5,vf,code);

call put_fparamacs(pstream,1,1,y,code);
call put_fparamacs(pstream,2,1,x,code);
return;

/*ISOTHERMAL FLASH UNIT CALCULATING ROUTINE */

iflash:entry(punit,parms,switch,error_switch);

      /* additional declarations */

dcl punit ptr;
dcl unit_ptr entry(ptr,fixed bin,ptr,bit(1));
dcl same_comps entry(ptr,ptr,bit(1));
dcl iset_parm entry(ptr,fixed bin,fixed bin,fixed bin,bit(1));
dcl no_of_ iterations;
dcl epsilon float bin(63);
dcl (pstream1,pstream2,pstream3) ptr;

/* PRELIMINARY CHECKS */

call unit_ptr(punit,1,pstream1,code);
call unit_ptr(punit,2,pstream2,code);
call unit_ptr(punit,3,pstream3,code);

/* DO ALL STREAMS HAVE THE SAME COMPONENTS ? */
call same_comps(pstream1,pstream2,code);
if code
  then error_switch='1'b;
call same_comps(pstream1,pstream3,code);
if code
  then error_switch='1'b;
if error_switch
  then do;
    put skip edit('IFLASH: COMPONENTS IN INLET AND OUTLET STREAMS NOT THE SAME')(a);
    return;
  end;

```

FIGURE 10.17 CONTINUED

```

/* GET INPUT VARIABLES */

call unit_ptr(punit,0,pparm,code);
call set_parm(pparm,1,t,vtype,code);
call set_parm(pparm,2,p,vtype,code);
call iset_parm(pparm,3,no_of_iterations,vtype,code);
call set_parm(pparm,4,epsilon,vtype,code);

call strm_ptr(pstream1,0,pparm,code);
call set_parm(pparm,3,f,vtype,code);
call set_fparms(pparm,0,1,z,vtv,code);

/* PERFORM THE CALCULATIONS */

call equil(z,vtv,t,p,x,y,vf,epsilon,no_of_iterations,code);
if code
  then do;
    error_switch="1"b;
    put skip edit("IFLASH: NO CONVERGENCE")(a);
    return;
  end;

/* STORE OUTPUT VARIABLES */
call strm_ptr(pstream2,0,pparm,code);
call put_parm(pparm,1,t,code);
call put_parm(pparm,2,p,code);
call put_parm(pparm,3,f*vf,code);
call put_parm(pparm,4,enthal(y,vtv,t,p,"1"b,code),code);
call put_parm(pparm,5,1.,code);
do i=0 to 1;
call put_fparms(pparm,0,1,y,code);
end;

call strm_ptr(pstream3,0,pparm,code);
call put_parm(pparm,1,t,code);
call put_parm(pparm,2,p,code);
call put_parm(pparm,3,f*(1-vf),code);
call put_parm(pparm,4,enthal(x,vtv,t,p,"0"b,code),code);
call put_parm(pparm,5,0.,code);
call put_fparms(pparm,0,1,x,code);
call put_fparms(pparm,2,1,x,code);
x(*)=0;
call put_fparms(pparm,3,1,1,x,code);
call put_fparms(pparm,2,2,1,x,code);

return;

end strmcals;

```

UNIT TYPE = distillation
 REFERENCE = Risorous distillation Thiele-Geddes

NUMBER OF UNIT PARAMETERS = 9

PARAMETER	DIMENSION	TYPE
1- no_of_plates	0	
2- plates_below_fee	0	
3- reflux_ratio	0	
4- condenser_duty	4	
5- reboiler_duty	4	
6- upper_temp_limit	1	
7- lower_temp_limit	1	
8- tolerance	0	
9- max_it	0	

NUMBER OF INLETS = 1

INLET CONNECTIONS	STREAM TYPE
1- feed	std

NUMBER OF OUTLETS = 2

OUTLET CONNECTIONS	STREAM TYPE
2- overhead	std
3- bottoms	std

PROCEDURE TO CALCULATE THE UNIT = distillation

MINIMUM NO OF ARGUMENTS = 0
 MAXIMUM NO OF ARGUMENTS = 1

NUMBER OF LEVELS OF CALCULATION = 1

VALUE STATUS CODES FOR LEVEL = 1

UNIT PARAMETERS	
PARAMETER	CODE
1- no_of_plates	7
2- plates_below_fee	7
3- reflux_ratio	7
4- condenser_duty	13
5- reboiler_duty	13
6- upper_temp_limit	7
7- lower_temp_limit	7
8- tolerance	7
9- max_it	7

FOR CONNECTION = feed

THIS CONNECTION IS REQUIRED

STREAM CONNECTED TO THIS CONNECTION = std

VALUE STATUS CODE OF ALL PARAMETERS OF ALL COMPONENTS

FLOWING IN THIS STREAM = 7

PHASE 0 (TOTAL STREAM)

PHASE PARAMETER CODE

1- t 13

2- p 7

3- f 7

4- h 13

5- vf 7

FLOW PARAMETER CODE

1- z 7

PHASE 1

FLOW PARAMETER CODE

1- w 15

PHASE 2

FLOW PARAMETER CODE

1- x 15

FOR CONNECTION = overhead

THIS CONNECTION IS REQUIRED

STREAM CONNECTED TO THIS CONNECTION = std

VALUE STATUS CODE OF ALL PARAMETERS OF ALL COMPONENTS

FLOWING IN THIS STREAM = 7

PHASE 0 (TOTAL STREAM)

PHASE PARAMETER CODE

1- t 5

2- p 13

3- f 7

4- h 13

5- vf 13

FLOW PARAMETER CODE

1- z 13

PHASE 1

FLOW PARAMETER CODE

1- w 13

PHASE 2

FLOW PARAMETER CODE

1- x 13

FOR CONNECTION = bottoms

THIS CONNECTION IS REQUIRED

STREAM CONNECTED TO THIS CONNECTION = std

VALUE STATUS CODE OF ALL PARAMETERS OF ALL COMPONENTS

FLOWING IN THIS STREAM = 7

PHASE 0 (TOTAL STREAM)

PHASE PARAMETER CODE

1- t 5

2- p 13

3- f 13

4- h 13

5- vf 13

FLOW PARAMETER CODE

1- z 13

PHASE 1

FLOW PARAMETER CODE

1- w 13

PHASE 2

FLOW PARAMETER CODE

1- x 13

- 4) Add. It adds two streams and determines the product stream. It has two levels of calculation. Level one is only for material balancing. Level two is for both material and heat balancing. For level two of calculation both inlet streams should be either vapor or liquid. The outlet stream temperature is determined by a heat balance. The output stream pressure is assumed to be equal to the minimum of the inlet stream pressures.
- 5) Splitter. It separates an input stream into two output streams, each having the same composition, temperature, and pressure as the input. The distribution factor which is the unit parameter specifies the flow rate of each output stream as a fraction of the flow rate of the input stream.
- 6) Convergence (convg). It is a psuedo unit type which is used for stream convergence. Streams to be converged are input and output to this unit. The test for convergence is based on the total flow rate and composition. In other words, the unit indicates convergence, if the following relations are true:

$$|(f_{in} - f_{out})/f_{in}| \leq \text{tolerance}$$

$$|(z_{in,i} - z_{out,i})/z_{in,i}| \leq \text{tolerance}$$

for all components

If the above test fails then the output stream is set to be equal to the input stream.

The calculating routine for this unit type can perform two levels of calculation. In level one, given the unit parameters (i.e. maximum number of iterations, tolerance, etc.), and inlet and outlet streams, it tests for convergence as described above. In level two it initializes the unit parameters to the default values.

Some of the above calculating routines call upon physical and thermodynamic property estimation routines, for various functions. Although these property estimation routines have been based on simplified assumptions regarding the thermodynamic behavior of liquid and vapor mixtures, they could easily be replaced with more realistic models without requiring any change in calculating routines. Table 10.1 lists the property estimation methods currently implemented for this TBS. There are two options for estimating vapor pressure: (1) Antoine's equation, and (2) Riedel's equation [140]. The template of the property estimation methods table which is shown in Figure 10.13 indicates that the user of this TBS is allowed to choose either of these methods at any time. If the user does not specify the method the default would be Antoine's equation. The templates of the TBS control information and dimension table are also shown in Figure 10.13.

A pre-defined function has been established for this TBS to demonstrate the implementation and application of such functions. The

Table 10.1 TBS-II Property Estimation Methods

Vapor Pressure

Two methods for estimating vapor pressures have been implemented:

1 - Antoine's equation:

$$\ln(\text{pvap}) = \text{anta} - \text{antb}/(t + \text{antc})$$

where vapor pressure is in millimeters of mercury and t in Kelvins.

2 - Riedel's equation [140]:

$$\ln(\text{pvap}/p_c) = A - B/t_r + C \ln(t_r) + D t_r^6$$

where:

$$t_r = t/t_c$$

$$A = -35Q$$

$$B = -36Q$$

$$C = 42Q + \alpha_c$$

$$D = -Q$$

$$Q = 0.0838(3.578 - \alpha_c)$$

$$\alpha_c = \frac{0.315W + \ln(p_c)}{0.0838W - \ln(t_b/t_c)}$$

$$W = -35 + \frac{36}{(t_b/t_c)} + 42 \ln(t_b/t_c) - (t_b/t_c)^6$$

K Value:

$$k_i = y_i/x_i = \text{pvap}_i/p$$

Table 10.1 (continued)Enthalpy

Basis: enthalpy of pure components at 298°K; vapor state is zero.

Molal enthalpy in vapor phase:

$$\text{ENTHALPY}_V = \int_{298}^t \text{cpvdt} + \sum_i \text{delhg}_i$$

Molal enthalpy in liquid phase:

$$\text{ENTHALPY}_L = \text{ENTHALPY}_V - \sum_i \text{hv}_i$$

where

$$\text{cpv} = \sum_i [\text{cpvapa}_i + (\text{cpvapb}_i)t^2 + (\text{cpvopc}_i)t^3 + (\text{cpvabd}_i)t^4]$$

Equilibrium Composition of a Mixture

If $\sum_i z_i k_i \leq 1$ mixture is at or below bubble point:

$$x_i = z_i, y_i = 0 \text{ for all } i$$

If $\sum_i z_i / k_i \leq 1$ mixture is at or above dew point:

$$y_i = z_i, x_i = 0 \text{ for all } i.$$

In the two-phase region:

$$\sum_i \frac{z_i(1 - k_i)}{1 - \text{vf}(1 - k_i)} = 0$$

Half interval method is used to find the root of the above equation, vf, for an accuracy of \pm tolerance. Once vf (vapor fraction) is found x's and y's are found as follows:

$$x_i = \frac{z_i}{1 + \text{vf} * (k_i - 1)}, y_i = k_i * x_i \quad \text{for all } i$$

Table 10.1 (continued)Average Heat Capacity

Average heat capacity in vapor phase:

$$CPAVG_V(T_1, T_2) = [ENTHALPY_V(T_2) - ENTHALPY_V(T_1)] / (T_2 - T_1)$$

Average heat capacity in liquid phase:

$$CPAVG_L(T_1, T_2) = [ENTHALPY_L(T_2) - ENTHALPY_L(T_1)] / (T_2 - T_1)$$

Bubble Point Temperature

$$F(T_B) = 1 - \sum_i z_i k_i(T_B) = 0.0$$

Half interval method is used to find the root of the above equation, bubble point temperature, for an accuracy of \pm tolerance.

Given the enthalpy of a liquid or vapor mixture find, its temperature:

$$ENTHALPY_{GIVEN} - ENTHALPY_V(T) = 0$$

or

$$ENTHALPY_{GIVEN} - ENTHALPY_L(T) = 0$$

Half interval method is used to find the root of the above equations, temperature, for an accuracy of \pm tolerance.

function is a least squares line fitting function whose template and calculating and evaluating routines are shown in Figure 10.15 and 10.19 respectively. The examples that follow demonstrate the application of this TBS.

Examples

a) The example presented here is selected from the unpublished class notes of Professor L.B. Evans [29] with his permission.

Simulate the steady state behavior of a three-stage flash separation process shown in Figure 10.20a. The feed stream is a mixture with the following compositions:

<u>Component</u>	<u>Mole Fraction</u>
N-Butane	.25
N-Pentane	.25
N-Hexane	.25
N-Heptane	.25

The feed flow rate is 100 kg moles per hour. The stage specifications are:

<u>Stage</u>	<u>Temperature</u>	<u>Pressure</u>
1	225°F	100 psia
2	185°F	50 psia
3	125°F	14.7 psia

The computational flowsheet for this process is shown in Figure 10.20b. Units "M1" and "M2" are for mixing the streams and unit "TEST" is for

FIGURE 10.19 TRS-II - FUNCTION CALCULATING AND EVALUATING ROUTINES

l1calc.pl1

05/12/78 1254.9 edt Fri

```

l1calc:proc (pfunc,pars,switch,error_switch);

    /* DECLARATIONS */

    dcl (pfunc,pars) ptr;
    dcl (switch,error_switch) bit(1);

    dcl setin entry(fixed bin,fixed bin);
    dcl setrn entry(float bin(63));
    dcl func_ptr entry(ptr,ptr,bit(1));
    dcl put_parm entry(ptr,fixed bin,float bin(63),bit(1));

    dcl i,n;
    dcl (x,y,sx,sx2,sy,sy2,sxy,a0,a1,r2) float bin(63);
    dcl p ptr;
    dcl code bit(1);

    /* CALCULATING ROUTINE */

    put skip edit("L1CALC: ENTER THE NUMBER OF DATA POINTS:")(a);
    call setin(n,99999);

    sx,sy,sx2,sy2,sxy=0;

    put skip edit("L1CALC: ENTER X AND Y OF EACH POINT:")(a);
    do i=1 to n;
    put skip edit(i,"")(x(5),f(5),a);
    call setrn(x);
    call setrn(y);
    sx=sx+x;
    sy=sy+y;
    sx2=sx2+x**2;
    sy2=sy2+y**2;
    sxy=sxy+x*y;
    end;

    a0=(sy*sx2-sx*sxy)/(n*sx2-sx**2);
    a1=(n*sxy-sx*sy)/(n*sx2-sx**2);
    r2=(n*sxy-sx*sy)**2/((n*sx2-sx**2)*(n*sy2-sy**2));

    /* STORE THE COEFFICIENTS */
    call func_ptr(pfunc,p,code);
    call put_parm(p,1,a0,code);
    call put_parm(p,2,a1,code);

    /* PRINT R SQUARE */
    put skip edit("R2=",r2)(x(5),a,e(22,14,14));

    return;

    /* EVALUATING ROUTINE */

l1eval:entry(pparm,ars,result);

    /* additional declarations */
    dcl pparm ptr;
    dcl ars(*) float bin(63);
    dcl result float bin(63);
    dcl set_parm entry(ptr,fixed bin,float bin(63),fixed bin,bit(1));
    dcl vtype;

    /* set the coefficients */
    call set_parm(pparm,1,a0,vtype,code);
    call set_parm(pparm,2,a1,vtype,code);

    /* evaluate the function */
    result=a0+a1*ars(1);

    return;

end l1calc;

```

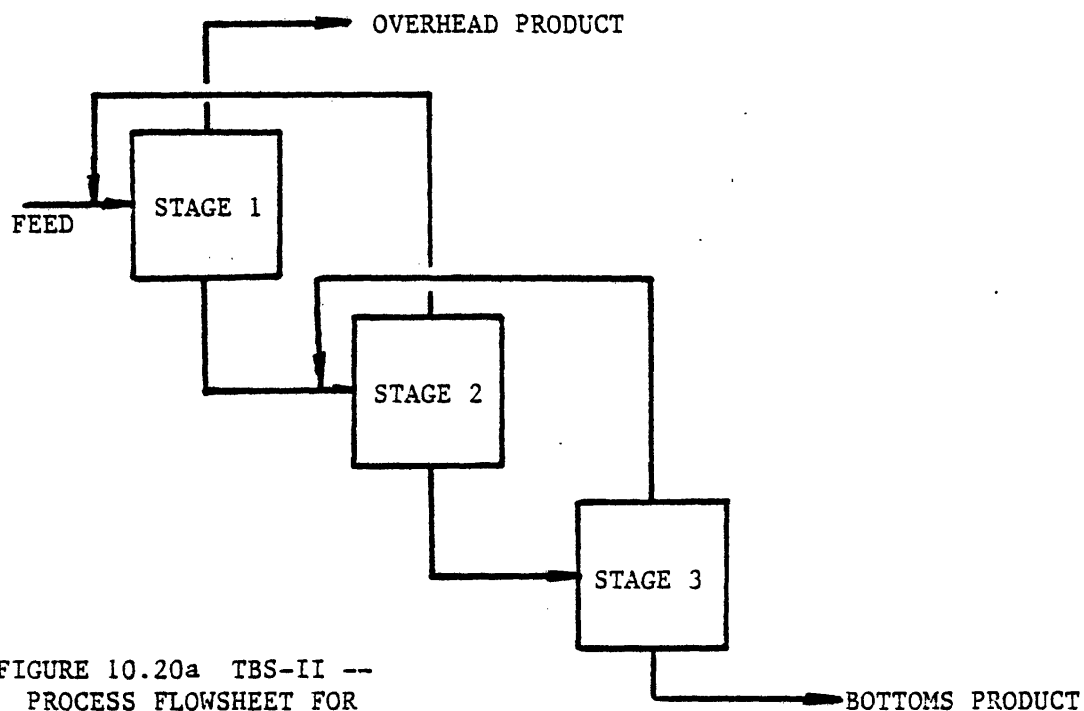


FIGURE 10.20a TBS-II --
PROCESS FLOWSHEET FOR
THE EXAMPLE CASE

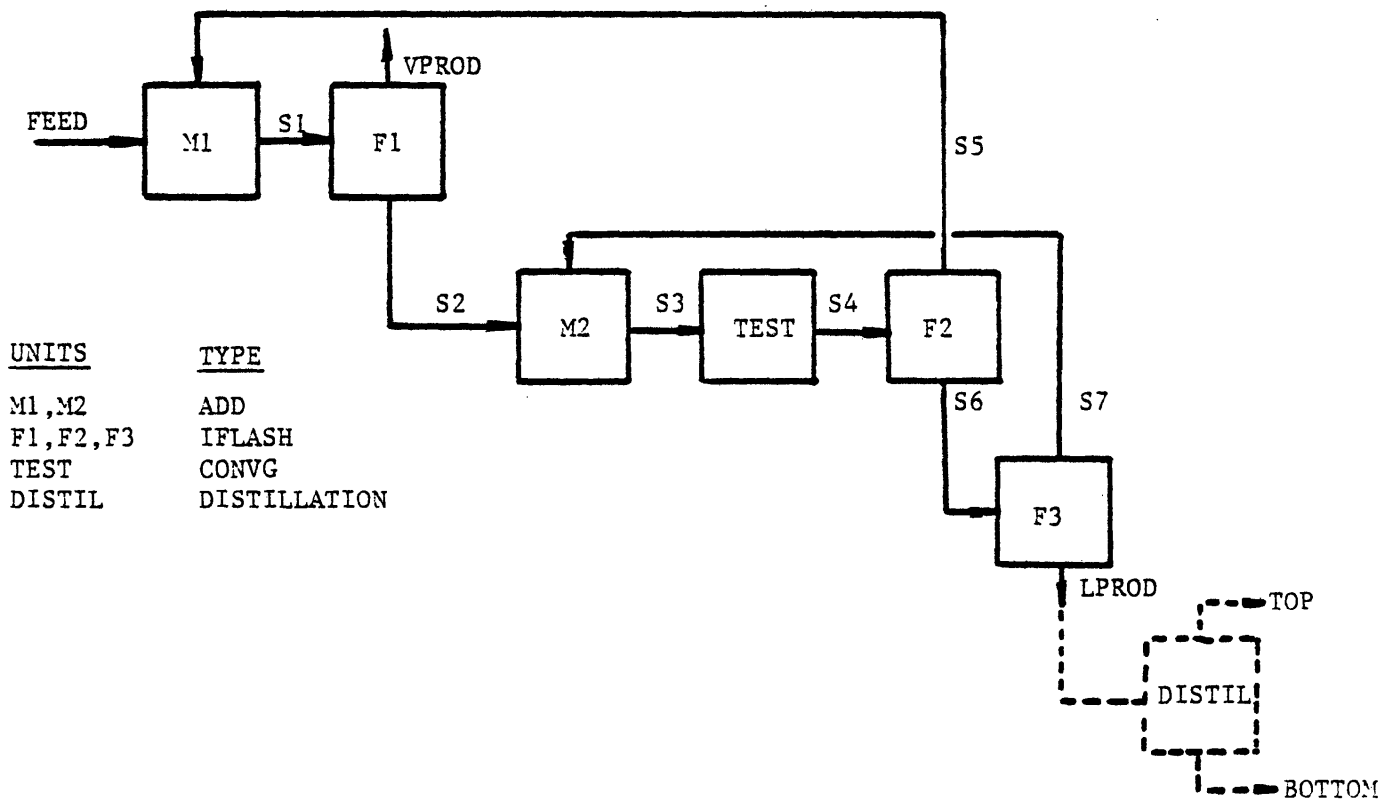


FIGURE 10.20b TBS-II -- COMPUTATIONAL FLOWSHEET FOR THE EXAMPLE CASE

stream convergence. The PEL computer session for simulating this flowsheet is shown in Figure 10.21.

b) Determine the sensitivity of the recovery of N-Pentane in the overhead and N-Hexane in the bottom product to variations in stage 2 pressure. The PEL computer session for this analysis is also shown in Figure 10.21.

c) Simulate the process with the following feed compositions:

<u>Component</u>	<u>Mole Fraction</u>
N-Butane	.2
I-Butane	.2
N-Pentane	.2
N-Hexane	.2
N-Heptane	.2

The PEL computer session for simulating this case is shown in Figure 10.22.

d) Distill the bottom product of the flowsheet described in part (a) by a distillation column with the following specifications:

Number of plates = 10

Plates below feed = 5

Reflux ratio = 2.5

Overhead flow rate = 30% of feed

Total condenser

Partial reboiler.

FIGURE 10.21 TBS-II - A PEL COMPUTER SESSION FOR THE EXAMPLE CASE

pel brief

```

*beginnings of attachment process.
**enter the name of TBS you wish to use now:TBS-II

*attachment process has been successful.

*new process created at:05/12/78 1342.6 edt Fri
**enter maximum number of components(0 to 200):5

**COMMAND :      $DESCRIBE THE PROCESS CONFIGURATIONS

**continue:create units(m1,m2) type=add;

**COMMAND :create units(f1,f2,f3) type=iflash;

**COMMAND :create unit(test) type=conv;

**COMMAND :      create streams(feed,1prod,vprod,s1,s2,s3,s4,s5,s6,s7);

**COMMAND :connect unit m1 at in1=feed in2=s5 out=s1;

**COMMAND :connect unit f1 at in=s1 overhead=vprod bottoms=s2;

**COMMAND :connect unit m2 at all=s2,s7,s3;

**COMMAND :connect unit test at all=s3,s4;

**COMMAND :connect unit f2 at all=s4,s5,s6;

**COMMAND :connect unit f3 at all=s6,s7,1prod;

**COMMAND :      $DESCRIBE THE COMPOSITIONS OF THE STREAMS

**continue:open component private file(data);

**enter the relative or absolute pathname of the directory containing the private component file 'data'
(if it is the same as your working directory ,>udd>ICPES>Arab-Ismaili, enter a null line):

*component private file data has been created at: 04/19/78 1032.9 est Wed
  by system: GPES          serial_no: 1 compat_level: 1
  and TBS: TBS-II         serial_no: 1 compat_level: 1
  and it has not been accessed by any other incompatible system since.
  remark:source: Reid and sherwood,Properties of liquids and Gases, 3rd ed.
**COMMAND :load components(nc4,nc5,nc6,nc7);

**COMMAND :create flow all all;

**COMMAND :specify stream(feed)(f=100);

**COMMAND :read flow(feed) all;

**enter the following flow      parameters:
stream feed
  phase      0
    component nc4
      1-z          :.25

    component nc5
      1-z          :.25

    component nc6
      1-z          :.25

    component nc7
      1-z          :.25

**COMMAND :

```

```

**continue:          $SPECIFY UNIT PARAMETERS$

**continue:read units all$

**enter the following unit      parameters:
unit test
  1-max_liter      :50
  2-n_liter        :0
  3-tolerance      :.01
  4-flas          :

unit f3
  1-tear           :125'f'
  2-pres           :14.7'psia'
  3-max_liter      :50
  4-tolerance      :.01

unit f2
  1-tear           :185'f'
  2-pres           :50'psia'
  3-max_liter      :50
  4-tolerance      :.01

unit f1
  1-tear           :225'f'
  2-pres           :100'psia'
  3-max_liter      :50
  4-tolerance      :.01

unit a2
  1-max_liter      :
  2-tolerancs     :

unit a1
  1-max_liter      :
  2-tolerancs     :

**COMMAND :          $ASSUME THE PARAMETERS OF THE TEAR STREAMS$

**continue:lets stream s4=feed$

**COMMAND :lets flow s4=feed$

**COMMAND :          $SIMULATE THE PROCESS$

**continue:calculate units(f2,f3,a1,f1,a2,test,(f2));

**enterins routine iflash      for level 1 calculation of unit 'f2'
**enterins routine iflash      for level 1 calculation of unit 'f3'
**enterins routine add         for level 1 calculation of unit 'a1'
**enterins routine iflash      for level 1 calculation of unit 'f1'
**enterins routine add         for level 1 calculation of unit 'a2'
**enterins routine convs       for level 1 calculation of unit 'test'
**enterins routine iflash      for level 1 calculation of unit 'f2'
**enterins routine iflash      for level 1 calculation of unit 'f3'
**enterins routine add         for level 1 calculation of unit 'a1'
**enterins routine iflash      for level 1 calculation of unit 'f1'
**enterins routine add         for level 1 calculation of unit 'a2'
**enterins routine convs       for level 1 calculation of unit 'test'
**enterins routine iflash      for level 1 calculation of unit 'f2'
**enterins routine iflash      for level 1 calculation of unit 'f3'
**enterins routine add         for level 1 calculation of unit 'a1'
**enterins routine iflash      for level 1 calculation of unit 'f1'
**enterins routine add         for level 1 calculation of unit 'a2'
**enterins routine convs       for level 1 calculation of unit 'test'
**enterins routine iflash      for level 1 calculation of unit 'f2'
**enterins routine iflash      for level 1 calculation of unit 'f3'
**enterins routine add         for level 1 calculation of unit 'a1'
**enterins routine iflash      for level 1 calculation of unit 'f1'
**enterins routine add         for level 1 calculation of unit 'a2'
**enterins routine convs       for level 1 calculation of unit 'test'
**COMMAND :#profile dflas=1$

**COMMAND :

```


FIGURE 10.21 CONTINUED

```

**continue:
*PRINT THE RESULTS
**continue:print v(s,vprod,p0,f), s,iprod,p0,f))
s,vprod,p0,f= 3.59738e+001 kmol/hr
s,iprod,p0,f= 6.32537e+001 kmol/hr
**COMMAND:print flow (vprod,iprod) parameters)
calculated calculated

```

Phase	Component	Value	Dimension	Status
Phase 0	component#4	5.70140e-001	value	calculated
	component#5	2.76137e-001	value	calculated
	component#6	1.08913e-001	value	calculated
	component#7	4.58074e-002	value	calculated
Phase 1	component#4	5.70140e-001	value	calculated
	component#5	2.76137e-001	value	calculated
	component#6	1.08913e-001	value	calculated
	component#7	4.58074e-002	value	calculated
Phase 2	component#4	0.00000e+000	value	calculated
	component#5	0.00000e+000	value	calculated
	component#6	0.00000e+000	value	calculated
	component#7	0.00000e+000	value	calculated
Flow of stream Iprod	component#4	0.00000e+000	value	calculated
	component#5	0.00000e+000	value	calculated
	component#6	0.00000e+000	value	calculated
	component#7	0.00000e+000	value	calculated
Phase 0	component#4	6.96361e-002	value	calculated
	component#5	2.30369e-001	value	calculated
	component#6	3.31335e-001	value	calculated
	component#7	3.68545e-001	value	calculated
Phase 1	component#4	6.96361e-002	value	calculated
	component#5	2.30369e-001	value	calculated
	component#6	3.31335e-001	value	calculated
	component#7	3.68545e-001	value	calculated
Phase 2	component#4	0.00000e+000	value	calculated
	component#5	0.00000e+000	value	calculated
	component#6	0.00000e+000	value	calculated
	component#7	0.00000e+000	value	calculated
Phase 2	component#4	6.96361e-002	value	calculated
	component#5	2.30369e-001	value	calculated
	component#6	3.31335e-001	value	calculated
	component#7	3.68545e-001	value	calculated

FIGURE 10.21 CONTINUED

```

**continue!          $SAVE THE PROCESS$

**continue!open Process file(demo)!

**enter the relative or absolute pathname of the directory containing the process file 'demo'
(if it is the same as your working directory ,>udd>ICPES>Arab-Ismaili, enter a null line)!

*Process file 'demo' is opened.
**COMMAND !save Process flash_seperation!

*Process has been saved.
**COMMAND :          $FART b      -   SENSITIVITY ANALYSIS $

**continue!repeat for u.f2.Pres from(30'Psia') to(70'Psia') by(10'Psia')!

**COMMAND !calculate units(f2,f3,m1,f1,m2,test(,f2))!

**COMMAND !specify variables (rnc5=s.vprod.P0.f*f.vprod.P0.nc5.z/(s.feed.P0.f*f.feed.P0.nc5.z)*100,
**continue!          rnc6=s.lprod.P0.f*f.lprod.P0.nc6.z/(s.feed.P0.f*f.feed.P0.nc6.z)*100,
**continue!          u.test.n_iter=0      )!

**COMMAND !print variables(u.f2.Pres,rnc5,rnc6)!

          u.f2.Pres=          2.04150e+000 atm          specified
          rnc5=          5.61943e+001
          rnc6=          7.39111e+001
**COMMAND !loop!

          u.f2.Pres=          2.72200e+000 atm          specified
          rnc5=          4.43336e+001
          rnc6=          8.39143e+001
          u.f2.Pres=          3.40250e+000 atm          specified
          rnc5=          4.12333e+001
          rnc6=          8.46989e+001
          u.f2.Pres=          4.08300e+000 atm          specified
          rnc5=          3.77734e+001
          rnc6=          8.41252e+001
          u.f2.Pres=          4.76350e+000 atm          specified
          rnc5=          3.60755e+001
          rnc6=          8.38169e+001
**COMMAND !end!

*thank you Arab-Ismaili for trying GPES come back soon, bye!
r 1420 15.788 332.606 5656

```

```

pel brief

*beginning of attachment process.
**enter the name of TBS you wish to use now:TBS-II

*attachment process has been successful.

*new process created at:05/12/78 1421.6 edt Fri
**enter maximum number of components(0 to 200):0

**COMMAND :          $LOAD THE PROCESS$

**continue:open process file(demo)

**enter the relative or absolute pathname of the directory containing the process file 'demo'
(if it is the same as your working directory ,>udd>ICPES>Arab-Isaaili, enter a null line):

*process file 'demo' is opened.
**COMMAND :load pr flash_seperation!

*process 'flash_seperation' has been created at:05/12/78 1342.6 edt Fri
  by system: GPES          serial_no: 1 compat_level: 1
  and TBS: TBS-II         serial_no: 1 compat_level: 1
  and it has not been accessed by any other incompatible system since.
*process has been loaded.
**COMMAND :          $SIMULATE FOR PART c OF EXAMPLE CASE$

**continue:open component private file(data)

**enter the relative or absolute pathname of the directory containing the private component file 'data'
(if it is the same as your working directory ,>udd>ICPES>Arab-Isaaili, enter a null line):

*component private file data has been created at: 04/19/78 1032.9 est Wed
  by system: GPES          serial_no: 1 compat_level: 1
  and TBS: TBS-II         serial_no: 1 compat_level: 1
  and it has not been accessed by any other incompatible system since.
  remark:source: Reid and sherwood,Properties of liquids and Gasses, 3rd ed.
**COMMAND :load component(ic4)

**COMMAND :create flow all (ic4)

**COMMAND :          $SPECIFY NEW COMPOSITIONS$

**continue:read flow(feed)all)

**enter the following flow      parameters:
stream feed
  phase      0
  component nc4
             1-z          :.2
  component nc5
             1-z          :.2
  component nc6
             1-z          :.2
  component nc7
             1-z          :.2
  component ic4
             1-z          :.2

**COMMAND :          $ASSUME PARAMETERS OF TEAR STREAMS$

**continue:leta stream s4=feed!

**COMMAND :leta flow s4=feed!

**COMMAND :          $SIMULATE THE PROCESS$

**continue:calc (f2,f3,m1,f1,m2,test(f2))

***ERROR*** s 56 'L' is an invalid command object for 'calc' command.
*command ignored.
**COMMAND :calc units(f2,f3,m1,f1,m2,test(f2))

*enter:ins routine iflash      for level 1 calculation of unit 'f2'
*enter:ins routine iflash      for level 1 calculation of unit 'f3'
*enter:ins routine add         for level 1 calculation of unit 'm1'
*enter:ins routine iflash      for level 1 calculation of unit 'f1'
*enter:ins routine add         for level 1 calculation of unit 'm2'
*enter:ins routine convs       for level 1 calculation of unit 'test'
*enter:ins routine iflash      for level 1 calculation of unit 'f2'
*enter:ins routine iflash      for level 1 calculation of unit 'f3'
*enter:ins routine add         for level 1 calculation of unit 'm1'
*enter:ins routine iflash      for level 1 calculation of unit 'f1'
*enter:ins routine add         for level 1 calculation of unit 'm2'
*enter:ins routine convs       for level 1 calculation of unit 'test'
*enter:ins routine iflash      for level 1 calculation of unit 'f2'
*enter:ins routine iflash      for level 1 calculation of unit 'f3'
*enter:ins routine add         for level 1 calculation of unit 'm1'
*enter:ins routine iflash      for level 1 calculation of unit 'f1'
*enter:ins routine add         for level 1 calculation of unit 'm2'
*enter:ins routine convs       for level 1 calculation of unit 'test'
*enter:ins routine iflash      for level 1 calculation of unit 'f2'
*enter:ins routine iflash      for level 1 calculation of unit 'f3'
*enter:ins routine add         for level 1 calculation of unit 'm1'
*enter:ins routine iflash      for level 1 calculation of unit 'f1'
*enter:ins routine add         for level 1 calculation of unit 'm2'
*enter:ins routine convs       for level 1 calculation of unit 'test'
**COMMAND :continue:iflash

```

**continue: \$PRINT THE RESULTS\$

**continue: \$print v(s.vprod,#0,f,s.lprod,#0,f);

s.vprod,#0,f= 6.00717e+001 ksmol/hr calculated
 s.lprod,#0,f= 3.97889e+001 ksmol/hr calculated

**COMMAND \$print flow(vprod,lprod);

flow of stream vprod

Phase	Component	Direction	Value	Dimension	Status
0	nc4	1-z	3.01688e-001	calculated	calculated
		1-z	3.01688e-001	calculated	calculated
	nc5	1-z	2.09747e-001	calculated	calculated
		1-z	2.09747e-001	calculated	calculated
	nc6	1-z	1.15032e-001	calculated	calculated
		1-z	1.15032e-001	calculated	calculated
	nc7	1-z	5.73200e-002	calculated	calculated
1-z		5.73200e-002	calculated	calculated	
1	nc4	1-y	3.01688e-001	calculated	calculated
		1-y	3.01688e-001	calculated	calculated
	nc5	1-y	2.09747e-001	calculated	calculated
		1-y	2.09747e-001	calculated	calculated
	nc6	1-y	1.15032e-001	calculated	calculated
		1-y	1.15032e-001	calculated	calculated
	nc7	1-y	5.73200e-002	calculated	calculated
1-y		5.73200e-002	calculated	calculated	
2	nc4	1-x	0.00000e+000	calculated	calculated
		1-x	0.00000e+000	calculated	calculated
	nc5	1-x	0.00000e+000	calculated	calculated
		1-x	0.00000e+000	calculated	calculated
	nc6	1-x	0.00000e+000	calculated	calculated
		1-x	0.00000e+000	calculated	calculated
	nc7	1-x	0.00000e+000	calculated	calculated
1-x		0.00000e+000	calculated	calculated	

flow of stream lprod

Phase	Component	Direction	Value	Dimension	Status
0	nc4	1-z	4.64010e-002	calculated	calculated
		1-z	4.64010e-002	calculated	calculated
	nc5	1-z	1.83312e-001	calculated	calculated
		1-z	1.83312e-001	calculated	calculated
	nc6	1-z	3.28559e-001	calculated	calculated
		1-z	3.28559e-001	calculated	calculated
	nc7	1-z	4.16061e-001	calculated	calculated
1-z		4.16061e-001	calculated	calculated	
1	nc4	1-y	0.00000e+000	calculated	calculated
		1-y	0.00000e+000	calculated	calculated
	nc5	1-y	0.00000e+000	calculated	calculated
		1-y	0.00000e+000	calculated	calculated
	nc6	1-y	0.00000e+000	calculated	calculated
		1-y	0.00000e+000	calculated	calculated
	nc7	1-y	0.00000e+000	calculated	calculated
1-y		0.00000e+000	calculated	calculated	
2	nc4	1-x	4.64010e-002	calculated	calculated
		1-x	4.64010e-002	calculated	calculated
	nc5	1-x	1.83312e-001	calculated	calculated
		1-x	1.83312e-001	calculated	calculated
	nc6	1-x	3.28559e-001	calculated	calculated
		1-x	3.28559e-001	calculated	calculated
	nc7	1-x	4.16061e-001	calculated	calculated
1-x		4.16061e-001	calculated	calculated	

FIGURE 10.22 CONTINUED

```

**continue:          $PART d - DISTILLATION $
**continue:load process flash_seperation;

*process 'flash_seperation' has been created at:05/12/78 1342.6 edt Fri
  by system: GPES          serial_no: 1 compat_level: 1
  and TBS: TBS-II         serial_no: 1 compat_level: 1
  and it has not been accessed by any other incompatible system since.
*process has been loaded.
**COMMAND :create unit(distil) type=distillation;

**COMMAND :connect unit distillation at feed=1*rod overhead=top bottoms=bottoms;

***ERROR*** s 27 'distillation' is an unknown unit
*command ignored.
**COMMAND :connect unit distil at feed=1*rod overhead=top bottoms=bottoms;

*stream 'top' does not exist. a stream of type 'std' has been created.
*stream 'bottoms' does not exist. a stream of type 'std' has been created.
**COMMAND :create flow (top,bottoms) all;

**COMMAND :read unit(distil)all;

#enter the following unit      Parameters:
unit      distil
  1-no_of_plates      :10
  2-plates_below_feed:5
  3-reflux_ratio      :2.5
  4-condenser_duty    :
  5-reboiler_duty     :
  6-upper_temp_limit:400'f'
  7-lower_temp_limit:50'f'
  8-tolerance         :.1
  9-max_it            :20

**COMMAND :specify stream(top) (f=.3*s.1*rod.p0.f);
**COMMAND :assume variables(s.top.p0.t=325,          s.bottoms.p0.t=375  );

**COMMAND :calc unit(distil);

DISTILLATION: ROUTINE NOT CONVERGED IN      20 ITERATIONS.
**COMMAND :print flow(top);

flow of stream top
  phase 0
    component=nc4
      parameter-----value_dimension-----status
      1-z                      2.32067e-001      calculated
    component=nc5
      parameter-----value_dimension-----status
      1-z                      7.12658e-001      calculated
    component=nc6
      parameter-----value_dimension-----status
      1-z                      5.44348e-002      calculated
    component=nc7
      parameter-----value_dimension-----status
      1-z                      8.42561e-004      calculated
  phase 1
    component=nc4
      parameter-----value_dimension-----status
      1-w                      0.00000e+000      calculated
    component=nc5
      parameter-----value_dimension-----status
      1-w                      0.00000e+000      calculated
    component=nc6
      parameter-----value_dimension-----status
      1-w                      0.00000e+000      calculated
    component=nc7
      parameter-----value_dimension-----status
      1-w                      0.00000e+000      calculated
  phase 2
    component=nc4
      parameter-----value_dimension-----status
      1-x                      2.32067e-001      calculated
    component=nc5
      parameter-----value_dimension-----status
      1-x                      7.12658e-001      calculated
    component=nc6
      parameter-----value_dimension-----status
      1-x                      5.44348e-002      calculated
    component=nc7
      parameter-----value_dimension-----status
      1-x                      8.42561e-004      calculated

```

FIGURE 10.22 CONTINUED

**continue! Print flow(bottoms);

flow of stream bottoms

Phase 0

component=nc4	parameter	value	dimension	status
	1-z	2.28183e-005		calculated
component=nc5	parameter			
	1-z	2.36730e-002		calculated
component=nc6	parameter			
	1-z	4.50006e-001		calculated
component=nc7	parameter			
	1-z	5.26131e-001		calculated

Phase 1

component=nc4	parameter	value	dimension	status
	1-y	0.00000e+000		calculated
component=nc5	parameter			
	1-y	0.00000e+000		calculated
component=nc6	parameter			
	1-y	0.00000e+000		calculated
component=nc7	parameter			
	1-y	0.00000e+000		calculated

Phase 2

component=nc4	parameter	value	dimension	status
	1-x	2.28183e-005		calculated
component=nc5	parameter			
	1-x	2.36730e-002		calculated
component=nc6	parameter			
	1-x	4.50006e-001		calculated
component=nc7	parameter			
	1-x	5.26131e-001		calculated

**COMMAND !save Process flash_seperation override;

*Process has been saved.

**COMMAND !end;

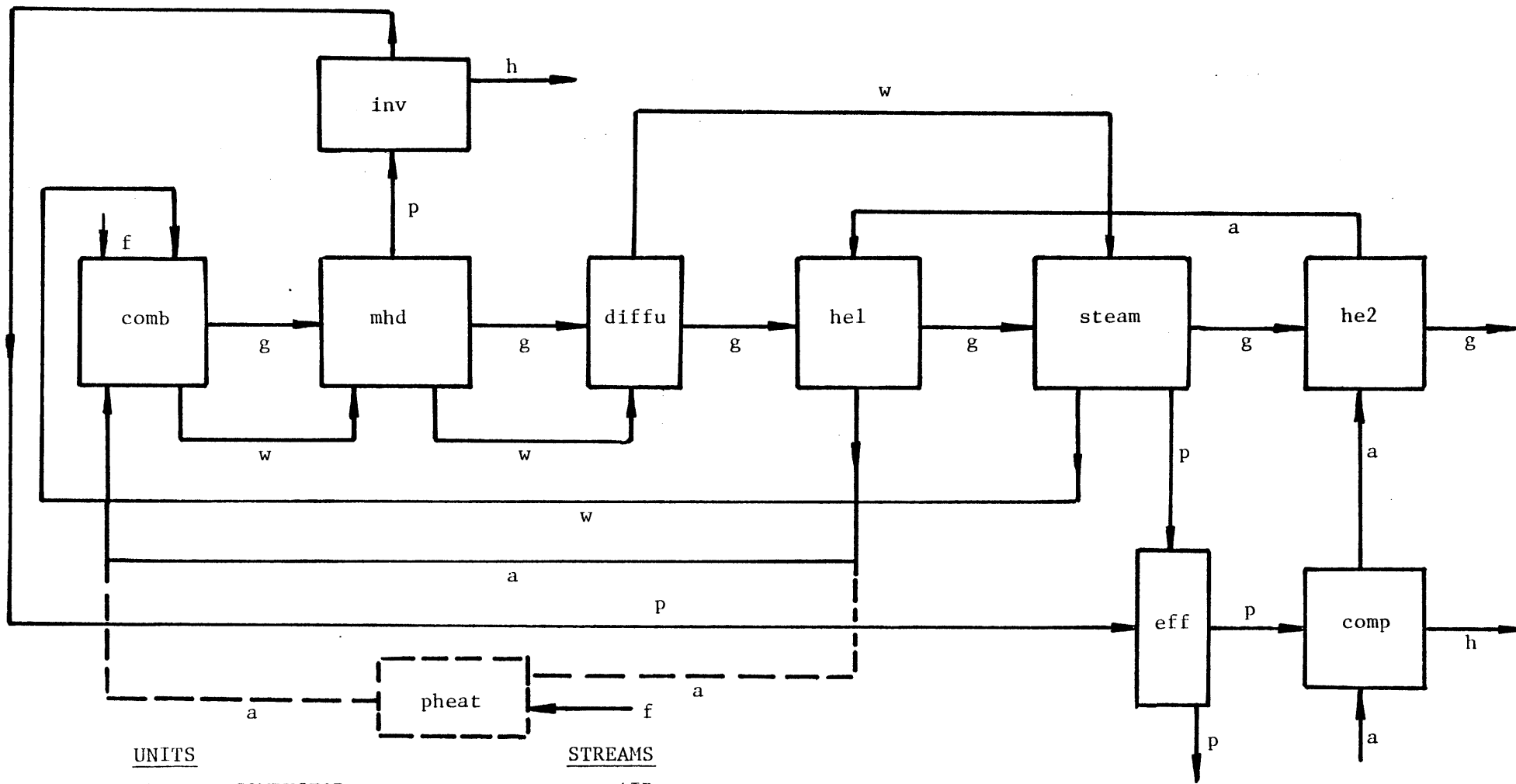
*thank you Arab-Ismaili for trying GPES come back soon, bye!

r 1454 10.546 191.693 3218

The PEL computer session for simulating this case is also shown in Figure 10.22.

10.4 A Prototype TBS for Magnetohydrodynamic (MHD) Processes

This TBS has been developed by Dr. H. Cohen and Mr. B. Misra from the MIT Energy Laboratory. The purpose of this study was to test the GPES independently, as well as to produce a useful system for the study of MHD power plants. The discussion about MHD power generators and detailed specification of this TBS is beyond the scope of this thesis and can be found elsewhere [90,91,92,93,94,95,96,97,98,99,100,101,102] . The process configuration of an open-cycle MHD-topped power plant is shown in Figure 10.23. This process has eight unit types and six stream types. The stream types are as follows: water, air, fuel, combustion gas, power, and heat. The unit types are as follows: combustor, MHD generator, inverter, diffuser, heat exchanger, steam plant, compressor, and efficiency. The last mentioned unit is only a pseudo-unit; it calculates the total power and the efficiency of the system. The combustor has two inlet streams: one for air, through which air at a prescribed pre-heat temperature is fed, and the other for fuel and seed. The combustion gas from the combustor is fed to the MHD generator where power is generated. The inverter serves to convert the power from the generator to A.C. power. The diffuser receives the combustion gas from the MHD generator and reduces the flow velocity to a prescribed value. The heat exchangers are used to heat the air to specified temperatures,



- UNITS
- comb - COMBUSTOR
 - comp - COMPRESSOR
 - diffu - DIFFUSER
 - eff - EFFICIENCY
 - he1,he2 - HEAT EXCHANGER
 - inv - INVERTER
 - mhd - MHD-GENERATOR
 - pheat - PRE-HEATER
 - steam - STEAM-PLANT

- STREAMS
- a - AIR
 - f - FUEL
 - g - COMBUSTION GAS
 - h - HEAT
 - p - POWER
 - w - WATER

FIGURE 10.23 MHD TBS -- PROCESS FLOWSHEET OF AN OPEN CYCLE MHD TOPPED PLANT

while the steam plant serves to generate power from the heat losses from the combustor, the MHD generator, the diffuser, and the enthalpy of combustion gas flowing within it. The compressor draws atmospheric air and feeds it to the heat exchanger II, after a specified compression. The power from the inverter and the steam plant is fed into the pseudo-unit efficiency, from which the compressor draws the requisite amount of power for its operation. It calculates the efficiency of the system as the ratio of the net power output and the total energy supply at the combustor. The application of this TBS will be illustrated by the following examples.

Examples:

The model for the process flowsheet shown in Figure 10.23 has already been saved.

a) Determine the sensitivity of the overall system efficiency to variations in condensation pressure at the termination of the steam cycle. The PEL computer session for this analysis is shown in Figure 10.24.

b) An externally fired pre-heater, as shown by the dashed lines in Figure 10.23 may be used to raise the temperature of the air coming out of the high temperature heat exchanger. Determine the effect of the degrees of pre-heat on the overall system efficiency. The PEL computer session for this case is shown in Figure 10.25.

FIGURE 10.24 MHD TBS - A FEL COMPUTER SESSION FOR THE EXAMPLE CASE

Fel brief

```

*beginning of attachment process.
**enter the name of TBS you wish to use now!MHD

*attachment process has been successful.

*new Process created at:05/07/78 2004.5 edt Sun
**enter maximum number of components(0 to 200):0

**COMMAND :      $---LOAD THE PROCESS---$

**continue:open Process file(misra);

**enter the relative or absolute pathname of the directory containing the Process file "misra"
(if it is the same as your working directory ,>udd>ICPES>HCohen, enter a null line)!

*Process file "misra" is opened.
**COMMAND !load Process demo;

*Process "demo" has been created at:04/19/78 0914.8 edt Wed
      by system: GPES          serial_no: 1 compat_level: 1
      and TBS: MHD            serial_no: 1 compat_level: 1
      and it has not been accessed by any other incompatible system since.
*Process has been loaded.
**COMMAND !      $---PART (a) SENSITIVITY OF OVERALL SYSTEM EFFICIENCY TO VARIATIONS IN CONDENSATION PRES.---$

**continue:repeat for u.steam.pes from(3000) to(10000) by(1000);

**COMMAND !calculate units(steam,eff);

*entering routine steam_inter      for level      1 calculation of unit      "steam"
*entering routine effc_inter       for level      1 calculation of unit      "eff"
**COMMAND !profile dflag=1;

**COMMAND !print variables(u.steam.pes,u.eff.effic);

      u.steam.pes=          3.0000e+003 newtons/sqm      specified
      u.eff.effic=         4.8425e-001                    calculated
**COMMAND !loop;

      u.steam.pes=          4.0000e+003 newtons/sqm      specified
      u.eff.effic=         4.8464e-001                    calculated
      u.steam.pes=          5.0000e+003 newtons/sqm      specified
      u.eff.effic=         4.8416e-001                    calculated
      u.steam.pes=          6.0000e+003 newtons/sqm      specified
      u.eff.effic=         4.8290e-001                    calculated
      u.steam.pes=          7.0000e+003 newtons/sqm      specified
      u.eff.effic=         4.8094e-001                    calculated
      u.steam.pes=          8.0000e+003 newtons/sqm      specified
      u.eff.effic=         4.7854e-001                    calculated
      u.steam.pes=          9.0000e+003 newtons/sqm      specified
      u.eff.effic=         4.7594e-001                    calculated
      u.steam.pes=          1.0000e+004 newtons/sqm      specified
      u.eff.effic=         4.7334e-001                    calculated
**COMMAND !

```

FIGURE 10.25 MHD TBS - ANOTHER FEL COMPUTER SESSION FOR THE EXAMPLE CASE

```

**COMMAND :      $---PART (b) - EFFECT OF AN EXTERNALLY FIRED PREHEATER ON PLANT EFFICIENCY---$
**continue:specify u(steam) (pes=5000);
**COMMAND :create unit(pheat) type=pre_heater;
**COMMAND :disconnect unit comb at          (airin);
**COMMAND :connect unit pheat at air_in=hel_air_out air_out=pheat_air_out power_in=ph_power;
*stream "pheat_air_out" does not exist. a stream of type "air" has been created.
**COMMAND :connect unit comb at air_in=hel_in pheat_air_out;
**COMMAND :specify unit(pheat)(eff=.7);
**COMMAND :repeat for s.pheat_air_out.p0.t from(199#00) to (2200) by(50);
**COMMAND :calc units(pheat,comb,mhd,inv,diffu,hel,steam,eff);
**COMMAND :specify variable (des_ext_pheat=s.pheat_air_out.p0.t - s.hel_air_out.p0.t);
**COMMAND :print variables(des_ext_pheat,u,eff,effic);
      des_ext_pheat=          0.0000e+000
      u,eff,effic=          4.8416e-001          calculated
**COMMAND :loop;
      des_ext_pheat=          5.0000e+001
      u,eff,effic=          5.0839e-001          calculated
      des_ext_pheat=          1.0000e+002
      u,eff,effic=          5.1181e-001          calculated
      des_ext_pheat=          1.5000e+002
      u,eff,effic=          5.0100e-001          calculated
      des_ext_pheat=          2.0000e+002
      u,eff,effic=          4.9426e-001          calculated
      des_ext_pheat=          2.5000e+002
      u,eff,effic=          5.2330e-001          calculated
      des_ext_pheat=          3.0000e+002
      u,eff,effic=          5.0368e-001          calculated
**COMMAND :end;

*thank you HCOhen for trying GFES come back soon, bye!
r 2200 58.637 840.057 14068

```

Chapter 11

RECOMMENDATIONS AND CONCLUSIONS11.1 Recommendations

There are many possible ways in which the GPES system could be extended to provide more user flexibility. Such possible extensions are discussed in this section, and are recommended for future work.

1) Drawing of Process Flowsheets

The system could be extended to draw a schematic diagram representing the process flowsheet in response to the user's command: print process flowsheet; the information required for this task is stored in the network of data structures representing the process. What is required is the development of an algorithm for this function.

2) Optional Units of Measurement for Printout Variables

The user may provide his input in any allowable units of measurement. The system automatically converts this input to the standard units of measurement for that parameter and stores the result in an appropriate location in the process network. Hence, the calculating routines always receive their input parameters in standard units, and should also compute their output parameters which are stored in the process network in standard units. When the user requests for output the parameters are also printed in standard units. It is desirable that the user be able to specify the units

of measurement for output parameters. The profile command can be extended to allow the user-system communication. For example, the following hypothetical command indicates that the printout of parameters having physical dimensions of temperature or pressure should be in degrees Centigrade and millimeters of mercury, respectively: `profile output_units(temperature = C, pressure = mmHg)`. The dimension table described in Chapter 4 contains the standard units of measurement for each physical dimension and other allowable options with their conversion factors for each dimension type. A temporary data structure is required to contain the current output units of measurement for each physical dimension. The entries of this data structure which are initially set to some default values or standard units should be updated in response to the profile commands specifying the output units. The print routines should use this data structure and the dimension table in converting the parameter values to the specified units of measurements before printing them.

3) Graph Plotting Capability

In sensitivity analysis and other parametric studies of a process flowsheet it is more desirable to present the results in graphs than tables. This capability could be incorporated to the system. The user could set up such graphs by the following hypothetical command:

```
graph graph_name initialize ( specification of graph
parameters );
```

where graph parameters could be attributes such as x axis title, y axis title, number of curves, character used in plotting each curve, maximum number of data points for each curve, length of x axis per unit of x, length of y axis per unit of y, origin, etc. The user could specify the data points on the curves by the following hypothetical command:

```
graph graph_name set (value or expression for x, value or
expression for y1, value or expression for y2, ...);
```

The graph can be plotted by the following hypothetical command:

```
graph graph_name print;
```

The following examples illustrates the application of such capability. In this example variations in steam plant efficiency are plotted versus the feed stream temperature.

```
graph A initialize (xtitle = "temperature," ytitle =
"efficiency," ncurves = 1, char = "*", maxn = 10, ...);
repeat for s.feed.po.t from (100) to (200) by (10);
calculate units (... , steam_plant, ...);
graph A set (s.feed.po.t, u.steam_plant.eff);
loop;
graph A print;
```

4) Merging of Two-Process Models

A user may save his process model in a process file and retrieve it later for further study. The user may also retrieve someone else's

process model with his permission. When retrieving a process the process in the file replaces the current process in the working area. This means that the process in the working area is lost if the user has not saved it before loading a new process. It is desirable to be able to merge two processes without having to build one over the other. This capability is especially useful when a team is analyzing a large process flowsheet and each member is studying a section of the flowsheet. It allows the assembly of all these sections to an integrated process model which enables the study of the whole process.

The user could request such an operation by a merge command similar to a load command.

Although the system could easily be extended to perform this function, its payoff may be offset by excessive computer overhead time. This is due to the effort required to make the two processes compatible. The following are some conflicts that should be resolved by the system before merging the two processes:

- a) Name Duplication.
- b) Addressability. The addressability of a process is by a pair of partition number and offset. Therefore in merging two processes, there will be address conflicts. This could be resolved by either of the following:
 - a. to relocate one of the processes

b. to change the system's memory management scheme from two-level addressability (partition number and offset) to a three-level addressability: process number, partition number and offset.

c) Merging of component directories. When merging two component directories the entry number of components in the new directory may not be the same as they were prior to merge. This change must be reflected in all streams referring to these components.

5) Choice of Calculating Routines

When a user requests for calculation the system will invoke the appropriate calculating routines specified in the templates. If a user wants to use different calculating routines, he would be able to do so by using some Multics commands. But this requires familiarity with the Multics system. Therefore it is recommended that the calculate commands and consequently the system itself be modified to allow the user to specify the calculating routines.

6) Extension of Templates

The function of calculating routines could be more simplified by allocating more types of error checking to the system. This requires extension of templates to convey the additional testing criteria to the system. Some of the examples of types of error checking that the system could be instructed to perform are as follows:

- a) same components should be present in inlet and outlet streams.
- b) upper and lower limits for parameter values.

Even the function of some calculating routines which represent simple models could be completely eliminated by describing them in templates and having the system carry out the calculations.

7) Billing and Statistics

It is desirable that the system be able to gather information regarding the usage of each user of each TBS, and the usage of each TBS by all users. This information may be used for billing, statistics, or other managerial purposes. As described in Chapter 8, provisions have been made in designing the TBS table and user's table for this information. Nevertheless, this information is not recorded. To record this information directly on these tables implies that the user should have write access to these files, which is not acceptable for security reasons. The other alternative is using the message segment facility of Multics. This involves the following:

- a) establish a message segment (a special file)
- b) modify the system so that when the user enters the system or leaves the system a message containing the required information would be sent to the above "safe box"
- c) modify the GPES administrative programs (update_tbs_tbl, update_user_tbl) to process these messages, update appropriate entries in the users' and TBS tables, and generate reports.

8) Unsteady State Analysis

The system is designed primarily for the study of steady-state operation of chemical processes. It is recommended that its application for unsteady state operations be explored.

9) Computational Schemes

The basic computational scheme of the system is sequential modular with user specifying the order of calculations. Although provisions have been made for implementing template-based systems with other computational schemes, the burden of such an effort is entirely on the TBS Administrators. Further research in this area is required to generalize and automate the process.

11.2 CONCLUSIONS AND THESIS CONTRIBUTIONS

There are two basic problems with the existing general purpose process simulators:

- a) The existing systems are applicable to those process flowsheets having only conventional vapor-liquid streams. This inflexibility makes it either impossible or very difficult to expand present-day systems to include other types of processes such as coal processing or electric power generating systems.
- b) The existing systems are mostly developed for simulation and do not provide the design atmosphere for process engineers. A general purpose simulator, to be effective as a design tool, must use a mode of operation, methods of input and output, and calculation techniques that minimize the effort required for designer-computer communication so as to maximize effective interaction between the designer and the computer. The time scale is important in process design.

The objective of this thesis was to develop a framework for the development of general purpose chemical process simulators that:

- a) are applicable to all types of chemical processes, and
- b) are more adaptable to the design environment.

In fulfillment of the above objective, the thesis has provided the following contributions in the field of computer aided design for process engineering:

1) Formulation of a general model for chemical processes, to represent the process flowsheet for any level of sophistication that may be used to analyze that process. The introduction of the concept of a general stream and flow parameters have been the major contributions for achieving this result.

2) Identification of the issues involved in the design of a computer system for process engineering. The work has distinguished two classes of problems:

- a) Those that are common to the design of any system,
- b) Those that are related to a particular system.

3) Based upon the above findings, the work has led to the design and implementation of a General Process Engineering System (GPES) which allows any group or organization to easily and systematically build its own system. These systems could be very simple or very sophisticated depending on the particular needs and applications of the group.

The design and implementation of the GPES as a tool for creating computer aided design systems for chemical process engineering provides the following advantages:

- a) Allows any group or organization to easily and systematically build its own design system, thereby reducing the time and effort required to produce such systems.
- b) The created system may not be limited to a particular class of processes. Such a system is open-ended and capable of analyzing any type of process.
- c) The created system is more adaptable to the design environment, and provides the user with features not usually found in existing general purpose simulators.

In summary, a system created by GPES (a TBS) has the following characteristics:

- a) Flexible. It is applicable for analyzing any type of process flowsheets. It is not limited to process flowsheets having only conventional vapor-liquid streams. The system can be easily modified, expanded and updated.
- b) The user communicates with the system by a problem oriented language (PEL). The user may enter his input data in any allowable units of measurement. Arithmetic expressions may be used where numerical data is expected. The system performs extensive types of error checking such as detection of over- or under-specification of process units and streams. The system produces over 200 easy-to-understand messages to detect the user's negligence.
- c) Interactive. Allowing the progressive design of a process.
- d) Integrated. Capable of the following functions:
 - i) Simulating or designing a process for any required level of sophistication such as: preliminary process feasibility studies, plant design, equipment sizing, plant modifications, debottlenecking studies, effects of operational changes on plants, contractor checkout. The appropriate programs should be provided in that TBS.
 - ii) Serving as a physical property data base system. It also promotes sharing of data among users.

- iii) Analyzing experimental data (regression analysis).
 - iv) Serving as a general purpose interpreter.
 - v) Serving as a desk calculator for arithmetic operations.
- e) Dynamic creation and modification of process flowsheet.
Enabling the user to instruct the computer to alter process configuration or operating parameters without having to redescribe the problem.
- f) The ability to save and retrieve user process models. This will save both designer and computer time in not having to reinitiate the problem. It also promotes sharing of process models among users, and enhances teamwork.
- g. It provides virtual memory. There is no limitation on the size of the process flowsheet being analyzed by the user.
- h. Ease of use in that no knowledge of programming and job control language is required. All that is required is knowledge of PEL (Process Engineering Language).
- 4) The work has provided a comprehensive example of the use of current systems programming techniques (structured programming, dynamic storage allocation, manipulation of arbitrary data structures, list processing, etc.) and current computer technology (time sharing, virtual memory, dynamic linking and loading) in a systems programming application of interest to chemical engineering.

Using the GPES several prototype template based systems have been created. The results of these efforts indicated that:

- a) The GPES allows the creation of computer systems for different types of processes.
- b) The creation of such systems is simple and systematic.
- c) The features provided by the system are very useful and desirable for simulation and design of chemical processes.

In conclusion, the above studies and findings, and the availability of such a GPES, will benefit the following groups:

- a) Those interested in computer aided design for chemical engineering applications.
- b) Those planning to implement their own computer aided design systems for chemical processes (by reducing the characteristic 20-100 man-years effort formerly required to produce such systems). It allows them to focus their effort on the chemical engineering side of the problem, which results in the development of better process modules and comprehensive physical and thermodynamic property calculation packages.
- c) The process designers, by providing the ideal creative environment for process design by computer.
- d) Chemical engineering students in process design courses. By allowing them to implement their own system or to use an already developed educational one. They would be able to study whole processes as carefully as we now study individual unit operations.

APPENDIX A

STATE OF THE ART

Process simulation is the representation of a chemical process by a mathematical model which is then solved to obtain information about the performance of the chemical process. The mathematical model is usually solved by a computer program. This computer program is generally known as a chemical process simulation program. The chemical engineer mostly deals with continuous deterministic system simulation. The simulation program may be a specific one prepared to simulate a particular process with a fixed plant layout or may be general to simulate any kind of process configuration. General purpose simulators use the modular approach. According to this method each chemical processing step is represented as a separate mathematical model called a unit module or building block or process unit. The unit modules are connected by data sets which represent the streams of materials and energy flowing between the units of the plant. An executive program supervises the information flow between the unit modules.

A simulation program may be used for steady-state or dynamic simulation. Although the aim of the process simulation program is to be a tool for design, most of the simulation programs work in the simulation/performance mode. It is characteristics of such a mode of calculation that all stream inputs and design parameters for the units are specified. The information flow in the simulation program is in the same direction as the heat and material flow in the chemical plant. Ideally, in the design mode of calculation the system inputs and/or design parameters are calculated from specified outputs. Design calculations can be

performed by iterative simulation. Iterative simulation may take place over an entire process with the user scheduling the cases to be computed, or it may take place internal to the simulation through the use of control blocks. Some process simulation programs perform optimization, equipment sizing or economic evaluation. In many cases, the connection between the programs which perform these tasks and the process simulation program is weak, frequently ad-hoc.

Since the steady-state process simulation program is the most widely used, the discussion is limited to this type of program. The important systems of this type that have been publicly acknowledged are listed in Table A.1. These systems have been developed by four types of organizations: chemical process companies, academic institutions, commercial computing services and consulting organizations, serving the chemical process industries. Evans [33] classifies these systems into two groups: first generation programs and second generation programs. He uses the terms "first and second generation" in an evolutionary sense, in that the development of the second generation programs has been based upon the technology gained by the development of the first generation programs. Kellogg's flexible flow sheet, Chevron's material and energy balancing, PACER and CHESS programs are examples of first generation systems and Monsanto's FLOWTRAN, Dupont's CPES, and Union Carbide's IPES are examples of the second generation systems.

Currently, a major research effort is underway at the Massachusetts Institute of Technology to develop a third generation computer system for chemical process engineering. The project, funded by the United States Department of Energy and budgeted at over three million dollars, is one of the greatest efforts of its kind. It is being directed by

Table A.1COMPUTER-AIDED PROCESS DESIGN SYSTEMS

<u>NAME</u>	<u>DEVELOPED BY</u>
ASPEN (Advanced System for Process Engineering)	Massachusetts Institute of Technology (under development) [3,4,5,6,7,8,9,10,30,31,32,33,74,151]
AGPSS	University of Michigan [5]
CAPEX (Computer-Aided Process Engineering System)	Chiyoda Chemical Engineering and Construction Co., Yokeshoma, Japan [106]
Chem E	Petroleum Consultants [131]
Chemical Process Simulator	Georgia Institute of Technology [5]
CHEMOS	University of British Columbia [5]
CHEOPS (Chemical Engineering Optimization System)	Shell Development Company [60]
CHESS (Chemical Engineering Simulation System)	University of Houston [115,116]
CHESS - 2 (commercial version of CHESS)	Chem Share, Inc.
CHEVRON (Generalized Heat and Material Balancing System)	Chevron Research Company [34,138]
CHIPS (Chemical Engineering Information Processing System)	Service Bureau Corp. [153]
CPES	Dupont
CONCEPT - III	Computer-Aided Design Center, Cambridge, U.K. [21]
COPE	Exxon
DESIGN	Chem Share, Inc.
DISCOSSA (Digital Simulation for Computation of Steady-state Analysis)	Oregon State University [28,86]
Extended U.P. PACER	University of Pennsylvania
Flexible Flowsheet	M.W. Kellogg Company [69,79,80,126]
FLOWPACK II	I.C.I., Ltd.

Table A.1 Continued

<u>NAME</u>	<u>DEVELOPED BY</u>
FLOWTRAN	Monsanto Chemical Company [149]
GEMCS	Canadian General Electric Co., McMaster University [22,72,73]
GEPDS (General Electric Process Design System)	General Electric Company [43]
GIFS (Generalized Interrelated Flow Simulation)	Service Bureau Corp. [25,152]
GPFS	Suntech, Inc. [5]
GPS	Phillips Petroleum/McDonnell Douglas Automation Company [5]
IPES	Union Carbide
MACSIM (version of PACER)	McMaster University
MAEBE (Material and Energy Balancing Execution)	University of Tennessee [78]
NETWORK	Imperial Chemical Co., Ltd. [1,12]
PACER (Process Assembly Case Evaluator Routine)	Purdue University and Dartmouth College [22,34,113,154,155]
PACER (MAD) (MAD version of PACER)	University of Houston [114]
PACER 245 (commercial version of PACER)	Digital Systems, Inc.
PDA Program 2249	McDonnell Douglas Automation Co. [5]
PEETPACK (Process Engineering Evaluation Techniques Package)	University of Aston, Birmingham, U.K. [128]
PRIMER	[162]
Process Analysis System	Oklahoma State University [5]
PROPS	University of Missouri-Rolla [5]
PROVES	[84,85]
QUIKBAL	Mobil

Table A.1 Continued

<u>NAME</u>	<u>DEVELOPED BY</u>
RUMBA	Kennecott Copper Corp. [5]
SEPSIM	University of Waterloo [5]
SIMUL-UNT	Universdad Nacional [5]
SLED (Simplified Language for Engineering Design)	University of Michigan [123]
SPAD	University of Wisconsin [5]
SPECS	Shell
SSI/100	Simulation Science, Inc. [5]
Symbol	Computer-Aided Design Center, Cambridge, U.K. [5]
Syntha II	Syntha Corp. of Greenwich/Control Data Corp. [5]
UOS (Unit Operations Simulator)	Bonner and Moore Engineering Associates [112]

Professor Lawrence B. Evans of MIT and Professor Warren D. Seider of the University of Pennsylvania. The system is named ASPEN (Advanced System for Process Engineering) and is expected to meet the needs of the Chemical Process Engineering profession in the 1980's. Those interested in additional information on this effort are directed to Evans et. al. [3,4,5,6,7,8,9,10,30,31,32,33,74,151].

Different aspects of the first and second generation systems are briefly discussed next. For an extensive review of the subject the reader is referred to Evans et. al. [34], who describe the first generation systems, and to Motard et. al. [117], who summarize recent work in steady-state process simulation.

A.1 Structure of the Programs

Evans [29] has classified the approaches used in the development of steady-state process simulation programs into three categories:

1. Development of a special-purpose computer program on an ad hoc basis.
2. Use of a set of subroutines as building blocks to develop a simulation of a process.
3. Use of a general-purpose executive program to simulate a particular class of processes.

The first approach requires developing the program from scratch. It requires the greatest programming effort and may be solved very efficiently using sophisticated numerical methods since the equations and constraints are fully defined prior to the effort of programming. However, it is inflexible when it comes to evaluating different plant layouts. This approach is almost never preferred.

In the second approach, the user writes his own main or executive routine which calls a number of subroutines to simulate individual units, to compute properties and to carry out other auxiliary functions. The structure of the flowsheet is incorporated into the user's program. This approach is, once again, not very flexible when it comes to evaluation of different plant layouts.

In the third approach, there exists an executive program which is available to the user. He simply has to provide a complete and unambiguous description of his problem in the form of a set of data cards or statements in a problem-oriented language (POL).

As noted by Motard, Shacham and Rosen [117], the general simulation programs using the last approach can be viewed as having one of two basic internal structures -- fixed structure or variable structure. In the former, the executive program is invariant with respect to flowsheet structure. The matching of streams to unit modules and the path of calculations through the program is determined by data supplied by the user.

In variable structure programs, a different executive program is constructed for each different flowsheet structure. Such programs employ a problem-oriented language which is interpreted directly into executable code (the compiler approach), or the POL may in turn generate a program in a procedure-oriented language, usually FORTRAN, which is then compiled, linked and executed (the pre-compiler approach). In the latter case, evaluation of different plant layouts can become quite expensive since the cost of compilation and linking is incurred every time. One disadvantage of the fixed structure executive programs is that they require all unit modules to be loaded into the computer memory regardless of whether a

particular unit module will actually be used. This can frequently lead to load modules whose size is greater than the available main memory.

A.2 Input Methods

Methods for inputting process specifications may vary from fixed format strings of words and numbers (e.g. Flexible Flowsheet, CHEVRON, PROVES) to free-format problem oriented languages. Process descriptions are provided in two forms. In the process oriented form, the user describes the contents, configuration, and data for his process independent of the calculation procedure. The user in later input or the program as part of its function will determine how to compute the desired output (e.g. PACER, CHEVRON).

In the calculation oriented form, the user describes the process as a sequence of calculations that may include recycle loops to be converged, each unit in the sequence receiving and transferring data to other units. The program performs calculations as specified (e.g. DESIGN, CHIPS, FLOWTRAN). This latter method gives the user control over calculations.

A.3 Data Checking

In most of the simulation programs the data checking must be done by the user, which is often not sufficient and more checking must be done by the simulator. It is often impossible to discover from the computer results that the source of some errors is due to absurd input data.

A.4 The Storage of the Data

The input data as well as calculated values are usually stored in arrays with fixed boundaries. In fixed structure programs, due to the limitation of the FORTRAN language, the storage space must always be allocated according to the maximum capacity of the simulation program without considering the actual problem size. Such use of storage space

reduces the flexibility and efficiency of these simulation programs. This problem may be solved partially by the use of main programs written by the user or POL generated main programs, since the boundaries of the arrays are declared according to the actual problem size. Another approach is that of using dynamic storage allocation which is provided by a language such as PL/1.

A.5 Operating Modes

Most systems have been designed to operate off-line, using card input. In some cases on-line operation via teletype was provided later although the system and inputs were fundamentally unchanged.

A.6 Available Unit Models

In general, the existing systems make available a library of standard unit modules for separation, heat exchange, mixing, and fluid transport, at varying levels of sophistication. Chemical reactors pose a problem, since general modeling is not available and most actual units are fairly specific.

A.7 Physical Property Determination

Thermodynamic or transport properties that are not specified directly must be calculated. This really requires integration of a simulation program with a properties calculator. Some systems have this feature and some systems do not, and the user is responsible for providing the necessary data.

A.8 Convergence Acceleration

The following convergence acceleration methods are being used in existing programs:

1. Successive Substitution,
2. Bounded Wegstein,

3. Dominant Eigenvalue,

4. Quasi-Newton.

Some systems use only one of these methods (e.g., FLOWTRAN, CONCEPT use Bounded Wegstein) while some other systems use more than one of these methods (CAPES uses Bounded Wegstein, Dominant Eigenvalue and Quasi-Newton).

APPENDIX B

TEMPLATE DEFINITION LANGUAGE

TDL is a simple command language by which the TBS Administrator instructs and communicates with "update_tdb" program to store or update the templates in the template data base. The basic elements of the language are commands. Each command is a request for an action to be taken by the "update_tdb" Program. The program accepts a command, interprets it, and finally accomplishes what the user intended by that command. Then the program is ready to accept another command. The program ignores and prints error messages for illegal commands. The program does not allow invalid data to be inserted. It prints messages and asks the user to reenter the appropriate data. The TDL command syntax is as follows:

```
command [object] [identifier]
```

The "object" and "identifier" are not always required. The "command" is a language keyword indicating the function of the command. The "object" is another language keyword indicating the object upon which the action should be taken. The "identifier" is either another language keyword, an integer number, or a user supplied identifier representing the type of the object. The complete list of TDL commands is given in Table B.1.

Permitted abbreviations of language keywords are given in Table B.2.

Reserved Words in TDL

The symbols "all", "none", ";", and symbols starting with the percent sign character (%) have special meanings in TDL and PEL. A unit, stream, and function type cannot be "all". A component type cannot be "all" or "none". The symbol "all" when used as an object type refers to all types of that object. Using the symbol "none" as a component type indicates that no component is allowed in that context. Parameter names and unit connection

TABLE B.1

THE TEMPLATE DEFINITION LANGUAGE COMMANDS

COMMAND	OBJECT	IDENTIFIER
1. insert	unit	"type"
	stream	"type"
	component	"type"
	function	"type"
	dimension	"number"
	property	"number"
2. delete	unit	"type"
	stream	"type"
	component	"type"
	function	"type"
	dimention	"number"
	property	"number"
3. replace	unitlevel	"type"
	streamlevel	"type"
	complevel	"type"
	funclevel	"type"
4. print	unit	"type"
	stream	"type"
	component	"type"
	function	"type"
	dimension	"number"
	property	"number"
	dimtable	----
	proptable	----
	ctlinfo	----
	vsctable	----
	all	----
5. list	units	----
	streams	----
	components	----
	functions	----
6. revise	ctlinfo	sysname
		serialno
		compatlevel
		defsdigit
		defddigit
		defdflag
		unitall
		streamall
	compall	
	funcall	
7. end	----	----

TABLE B.2

PERMITTED ABBREVIATIONS IN THE TDL COMMANDS

<u>KEYWORD</u>	<u>ABBREVIATION</u>
all	--
compall	ca
compatlevel	clvl
complevel	cl
component	comp, c
components	comps, cs
ctlinfo	ci
defddigit	dd
defdflag	ddf
defsdigit	ds
delete	d
dimension	dim, d
dimtable	dtbl, dt
end	--
funcall	fna
funclevel	fnl
function	func, fn
functions	funcs, fns
insert	i
list	l
print	p
property	prop, p
proptable	ptbl, pt
replace	r
revise	rvs
serialno	sno
stream	s
streamall	sa
streamlevel	sl
streams	strms, ss
sysname	sysn
unit	u
unitall	ua
unitlevel	ul
units	us
vsctable	vsctbl

names cannot be "all" or symbols starting with the percent sign character (%). This restriction has been imposed to provide greater user flexibility in referring to parameters and connections in PEL. In PEL a user may refer to a parameter either by its name or by the symbol %parameter_n (or %parm_n, or %pn), where n is the parameter number. Similarly, a unit connection position is referred either by its name or by %cnctn (or %cn), where n is the connection number. Also in PEL, the symbol "all" is used to refer to all parameters or all unit connections. Another reserved word in TDL is the character semicolon (;). Using a ";" as a routine's name indicates that no routine is implemented. There is no other reserved word in TDL. All symbols except the reserved words mentioned in this section can be used in any context. All symbols used in TDL cannot be more than 16 characters.

Program Interrupt

During the TDL session, the user can exit from the program by pushing the "QUIT" button on the terminal. This mechanism can be used for aborting the printout of a print command, or, in the case of insert or replace commands, for avoiding the need to enter additional information when the user has detected some errors in previously supplied information. Reentry after this abnormal exit can be accomplished by using the "pi" (Program Interrupt) command of Multics. After the reentry the current command activities will be ignored and the program is ready to accept a new command.

Use of this mechanism for cases other than those mentioned above may damage the data base, and should not be attempted.

TDL Commands

This section contains descriptions of TDL commands. The commands are presented in alphabetical order.

Delete Commands

These commands are used to delete the templates. They are as follows:

delete unit "type"

If the specified unit template exists, it will be deleted.

delete stream "type"

If the specified stream template exists, it will be deleted.

delete component "type"

If the specified component template exists, it will be deleted.

delete function "type"

If the specified function template exists, it will be deleted.

delete dimension "number"

The corresponding entry of the dimension table will be deleted.

delete property "number"

The corresponding entry of the property estimation table will be deleted.

End Command

This command is used to terminate the session. The program checks the consistency of the data base and will print any detected inconsistency. The user is then asked if he or she would like to exit. If the answer is "yes", the program asks for the default types of unit, stream, component, and function and these are made the first members of their respective categories. Clearly the above inquiry only takes place for those categories which contain more than one member. When the GPES Executive wants a default type, it looks for the first member of the corresponding directory.

Insert Commands

These commands are used to insert the templates into the template data base. The program prompts the user for all the required information. The program then asks the user whether the template is to be inserted into the template data base, or to be ignored. These commands are as follows:

insert unit "type"

This command is used to define a new unit type. The template of the specified unit type should not already exist. The program prompts the user for the following information:

- * Reference Information (48 characters).
- * Number of Unit Parameter (0 or a positive number).
- * Name and Dimension type of each parameter.
- * Number of inlets (Zero or a positive number).
- * Number of connections (A number not less than the above number).
- * Connection name and stream type for each connection. Using the symbol "all" as the stream type indicates that any type of stream could be connected to the unit. Otherwise, the specified stream type should have been already defined.
- * Procedure to calculate the unit. This should be the name of the calculating routine. If such a routine is not required or implemented, a semicolon (;) should be used.

If the user's response is not a ";", the program prompts the user for the following additional information regarding the calculating routine:

- * Minimum number of arguments (zero or a positive number).
- * Maximum number of arguments. A negative number can be used to indicate that the maximum number of arguments is unlimited. Otherwise

a positive number equal to or greater than the minimum number of arguments should be provided.

* Number of Levels of calculation (a positive number).

For each level of calculation, the program asks for the value status code of each unit parameter and information regarding the inlet and outlet streams. The latter consists of the following information for each unit connection:

* Connection status which should be 1, 2, or 3. The connection status of 1 indicates that a stream must be connected to the connection before the unit calculating routine could perform the specified level of calculation. The connection status of 2 indicates that such a stream is not permitted. The connection status of 3 indicates that such a stream is optional.

If the connection status is 1 or 3 and the stream type is not "all", the following information would also be required:

* If the stream can have any component, a value status code for all parameters of all components flowing in the stream.

* A value status code for each phase and flow parameter of each phase.

insert stream "type"

This command is used to define a new stream type. The template for the specified stream type should not already exist. The program prompts the user for the following information:

* Reference Information (up to 48 characters).

* Type of components flowing in the stream. Providing the symbol "none" in this context indicates that no component is allowed to flow in the stream. Providing the symbol "all" is the indication that any type of component may flow in the stream. Providing any

other symbol is interpreted as allowing only the specified type of components to flow in the stream.

- * Number of phases. (Zero or a positive integer number).

The program asks for the following information for each phase:

- * The number of phase parameters (0 or a positive integer number).
- * Name and dimension type of each phase parameter.
- * The number of flow parameters (Zero or a positive integer number).
For a stream with no component (Component type = none) this number should be zero.
- * Name and dimension type of each flow parameter.
- * Procedure to calculate the stream (Name of the calculating routine or a ";".)

If the user's response is not a ";", the program prompts the user for the following additional information regarding the calculating routine:

- * Minimum number of arguments (Zero or a positive integer number)
- * Maximum number of arguments. A negative number may be used to indicate that the maximum number of arguments is unlimited.
Otherwise a positive number equal to or greater than the minimum number of arguments should be provided.
- * Number of levels of calculations (a positive integer number).

The program also asks for the following information for each level of calculation:

- * If the stream can have any component, a value status code for all parameters of all components flowing in the stream.
- * A value status code for each phase and flow parameter of each phase.

Remarks

A unit template includes information about its inlet and outlet stream types. Therefore, all such streams should be defined prior to the definition of the unit. Updating these stream templates after the unit template has been defined will impose some restrictions. To update a template one has to delete the template and insert a new one (except for replacing information regarding a level of calculation). To insert a stream type which is already rooted in the existing unit templates, the program forces the user to use the same number of phases as before, and for each phase the same number of phase and flow parameters as before. Therefore if the above restrictions will prevent the desired updatings, the user has to delete the templates of all units referring to that stream type. Once the stream has been updated, the unit templates which have been deleted should be reinserted.

insert component "type"

This command is used to define a new component type. The template of the specified component type should not already exist. The program prompts the user for the following information:

- * Reference information (up to 48 characteres).
- * Number of component parameters (Zero or a positive integer number).
- * Name and dimension type of each parameter.
- * Procedure to calculate the component (name of the calculating routine of a ";").

If the user's response is not a ";", the program prompts the user for the following information regarding the calculating routine:

- * Minimum number of arguments (Zero or a positive integer number).

- * Maximum number of arguments. A negative number may be used to indicate that the maximum number of arguments is unlimited. Otherwise a positive integer number equal to or greater than the minimum number of arguments should be provided.
- * Number of levels of calculation (a positive integer number).
- * A value status code for each component parameter for each level of calculation.

All component types mentioned in defining the stream templates should be defined prior to terminating the session.

insert function "type"

This command is used to define a new function type. The template for the specified function type should not already exist. The program prompts the user for the following information:

- * Reference information (up to 48 characters).
- * Number of function parameters (Zero or a positive integer number).
- * Name and dimension type of each parameter.
- * Procedure to evaluate the function, or a ";".
- * Number of arguments for the above routine (if it is not a ";").
- * Procedure to calculate the function (name of the calculating routine or a ";").

If the user's response is not a ";", the user is prompted for the following additional information:

- * Minimum number of arguments. A negative number may be used to indicate that the maximum number of arguments is unlimited. Otherwise a positive integer number equal to or greater than the minimum number of arguments should be provided.
- * Number of levels of calculation (a positive integer number).

- * A value status code for each function parameter for each level of calculation.

insert dimension "number"

This command is used to define a dimension type. A dimension type is either zero or a positive integer number. The dimension type of zero is always represents the dimensionless parameters and hence, should not be defined.

When defining the other dimension types, the program prompts the user for the following information:

- * The name of the physical dimension (e.g., temperature, pressure, etc.).
- * The standard units of measurement. Units of measurement should not contain any apostrophes.
- * The number of options (Zero or a positive integer number).
- * For each option the unit of measurement and conversion factors A and B. Conversion factors are used to convert a value given in an optional units of measurement to the standard units as follows:
value in standard units = A+B (value in optional units).

Each entry of the dimension table which represents a dimension type should be defined using this command. The sequence of defining each dimension type is not important, but before the session is terminated, all dimension types mentioned in other templates should have been defined and there should be no undefined dimension number (type) less than the maximum defined number. The last restriction is to ensure no entry of the dimension table is empty.

insert property "number"

This command is used to define each entry of the property estimation table. The program prompts the user for the following information:

- * The name of the property as it is to be known by the users.
- * The number of options available for estimating the property (a positive integer number).
- * The default option (a positive integer number not greater than the above number).

The sequence of defining each entry of the table is not important, but before the session is terminated, all the entries of the table should have been defined. In other words, there should be no undefined property number less than the maximum defined number.

List Commands

These commands are used to list the existing types of units, streams, components, or pre-defined functions. These commands are as follows:

list units

list streams

list components

list functions

Print Commands

These commands are used to print the templates. The specified templates should exit.

Print commands are as follows:

print unit "type"

It will print the specified unit template.

print stream "type"

It will print the specified stream template.

print component "type"

It will print the specified component template.

print function "type"

It will print the specified function template.

print dimension "number"

It will print the specified dimension type.

print property "number"

It will print the specified property.

print dimtable

It will print the entire dimension table.

print protable

It will print the entire property table.

print ctlinf

It will print the control information.

print vsctable

It will print the value status codes table.

The Value Status Codes Table, unlike the Dimension Table or Property Estimation Table, is not defined by the user (TBS Administrator).

It is a table describing the meaning of each 15 value status codes.

print all

It prints the entire template data base. The printout resulting from this command can be used as a reference guide for the users of the TBS.

The printout consists of the following:

- a) The System Control Information
- b) The list of Unit Types

- c) The list of Stream Types
- d) The List of Component Types
- e) The List of Function Types
- f) The Dimension Table
- g) The Property Estimation Table
- h) The Value Status Codes Table
- i) The Template for each Stream Type
- j) The Template for each Component Type
- k) The Template for each Unit Type
- l) The Template for each Function Type

Replace Commands

These commands are used to replace a part of a template associated with any one level of calculation. The program prompts the user for all the required information. The program then asks the user whether the given information is correct or not. If the new set of information is correct, it will replace the old set. These commands are as follows:

replace unit "type"

If the unit type exists and there is a calculating routine associated with it, the program prompts the user for the level of calculation to be replaced. If the above number is a positive integer number less than or equal to the number of levels of calculation for that unit type, the user is prompted for the required information regarding the unit calculating routine for the specified level of calculation. This information is exactly the same as the user provides for each level of calculation when inserting a new unit type.

replace stream "type"

If the stream "type" exists and there is a calculating routine associated with it, the program asks the user for the level of calculation to be replaced (a positive integer number).

If the input number is less than or equal to the number of levels of calculation for that stream type, the user is prompted for the required information regarding the stream calculating routine for the specified level of calculation. This information is exactly the same as the user provides for each level of calculation when inserting a new stream type.

replace component "type"

If the specified component template exists and a calculation routine is associated with it, the user is asked for the level of calculation to be replaced (a positive integer number). If the input number is not greater than the component's number of levels of calculation, then the user will be prompted for a value status code for each component parameter.

replace function "type"

If the specified function template exists and a calculating routine is associated with it, the user is asked for the level of calculation to be replaced (a positive integer number). If the input number is not greater than the function's number of levels of calculation, the user will be prompted for a value status code for each function parameter.

Revise Commands

These commands are used to change an item of the control information. These items can never be deleted, only changed. The program prompts the

user for all the required information. When a new template data base is created, the items of control information are initialized as follows:

```

sysname = ";"
serialno = 1
compatlevel = 1
defsdigit = 14
defddigit = 14
defdflag = 0
Procedure to calculate all units = ";"
Procedure to calculate all components = ";"
Procedure to calculate all functions = ";"
Procedure to calculate all streams = ";"

```

Note that ";" indicates that the routine is not implemented. The user should use the revise command to provide a name for the TBS (sysname) or to change the above default values. Revise commands are as follows:

revise ctlinfo sysname

The program prompts the user for the name of the TBS.

revise ctlinfo serialno

The program prompts the user for the TBS Serial Number (a positive integer number). The serial number should be revised for each generation of the TBS.

revise ctlinfo compatlevel

The program prompts the user for the TBS compatibility level (a positive integer number). Compatibility level should be revised whenever the updated version of the TBS is not compatible with previous generations. Two generations of a TBS are said to be incompatible if the processes created by each are not compatible. The problem arises when a process

created and saved under the control of one generation of the TBS is retrieved under the control of another generation of the TBS. Extending a TBS (adding new templates) will not make the TBS incompatible. Changing a template may or may not make the TBS incompatible. The following are some examples of actions that will make the TBS incompatible:

- i) Deleting an already defined template.
- ii) Changing the number of parameters of a unit, stream, component, or function.
- iii) Changing the number of phases of a stream.
- iv) Changing the number of connections of a unit.
- v) Changing the interpretation of a property number (type) or changing the number of properties in the Property Estimation Table.

When a user is loading an incompatible process, the GPES Executive will inform the user and ask him/her if the operation is to be continued. Therefore, it is recommended that the users be informed about the changes which have been made in the TBS so that they may decide whether or not those changes should be of any concern to them.

revise ctlinfo defsdigit

The program prompts the user for the default number of significant digits (between 1 and 14) to be used in printing the numerical data in response to PEL print commands.

revise ctlinfo defddigit

The program prompts the user for the default number of decimal digits (between 1 and 14) to be used in printing the numerical data in response to PEL print commands. This number should not be greater than the defsdigit.

revise ctlinfo defdflag

The program prompts the user for the default debugging flag (0 or 1). The flag of 0 which is for normal operation is recommended.

revise ctlinfo unitall

The program prompts the user for the name of the routine to calculate all units. If the user's response is not a ";", the user also has to provide the reference information and number of levels of calculation.

revise ctlinfo streamall

The program prompts the user for the name of the routine to calculate all streams. If the user's response is not a ";", the reference information and number of levels of calculation should also be provided.

revise ctlinfo compall

The program prompts the user for the name of the routine to calculate all components. If the user's response is not a ";", the reference information and number of levels of calculation should also be provided.

revise ctlinfo funcall

The program prompts the user for the name of the routine to calculate all functions. If the user's response is not a ";", the reference information and number of levels of calculation should also be provided.

APPENDIX C

TBS SERVICE ROUTINES

Service routines assist TBS Programmers in performing various tasks.

Service routines may be classified to the following six groups:

1. Routines performing basic operations. Given a pointer to a data structure, these routines return pointers to the related data structures.
2. Routines checking for existence of some components in a given stream.
3. Routines retrieving or storing the values of parameters from or into the process network. 78 out of 101 existing service routines fall into this group. Although using only 2 of these routines (GET_PARM and PUT_PARM) in conjunction with the basic service routines (first group) will be sufficient for accessing the parameters, the remaining 76 routines are provided for greater user flexibility and convenience. They may be used to access the parameters directly or indirectly; or access one parameter or a vector of parameters at a time. They may view the parameters of having different attributes.
4. Routines retrieving other variables of interest. Two routines fall in this group. One returns the value of a user supplied argument, and another one returns the method currently being used to estimate a physical property.
5. Routines directly interacting with the user. These routines are called to receive input data not provided in the process network, directly from the user.

6. Routines performing arithmetic operations on two parameter sets.

These routines are helpful in writing calculating routines for separation processes and material balancing in general.

A description of each service routine in each group follows next.

C.1 Basic Service Routines

Spointer

Given a pair of partition number and offset, it returns a pointer.

Usage

```
DECLARE SPOINTER ENTRY (FIXED BIN, OFFSET) RETURNS (PTR);
```

```
P = SPOINTER (PART_NO, OFF);
```

where:

P is the returned pointer (output)

PART_NO is the partition number (Input)

OFF is the offset (input)

Notes

P is null for the following cases:

- a. PART_NO is zero.
- b. PART_NO is invalid (negative or greater than the number of partitions).

UNIT PTR

Given the unit pointer, it returns the pointer either to the unit parameters or the stream connected at a given connection.

Usage

```
DCL UNIT_PTR ENTRY (PTR, FIXED BIN, PTR, BIT (1));
```

```
CALL UNIT_PTR (PUNIT, NUMBER, P, CODE);
```

where:

PUNIT is the pointer to the unit structure (input).

NUMBER is zero or connection number (input).

P is the pointer to the unit parameters structure, if NUMBER = 0. Otherwise, it is the pointer to the stream connected at the given connection (output).

CODE is the error code (output).

Notes

CODE is turned "on" and P is null for the following cases:

- a. PUNIT is null.
- b. NUMBER is negative or greater than the number of unit connections.
- c. The unit is not connected at the given connection.

Example

See Figure C.1.

GET COMP INDEX

It returns the component index (entry number in the Component Directory) of a given component.

Usage

```
DCL GET_COMP_INDEX ENTRY (CHAR (16), FIXED BIN, BIT (1));
CALL GET_COMP_INDEX (NAME, COMP_INDEX, CODE);
```

where:

NAME is the component's name (input).

COMP_INDEX is the component's index (output).

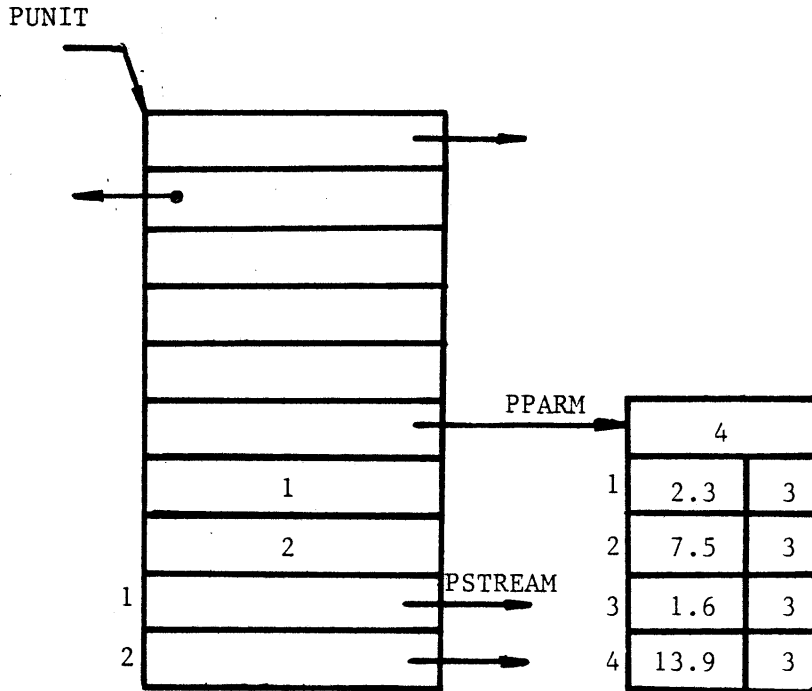
CODE is the Error Code (output).

Notes

CODE is turned "on" and COMP_INDEX is zero if the component does not exist.

Example

See Figure C.2.

UNIT_PTR

```
CALL UNIT_PTR (PUNIT,0,PPARM,CODE);
```

```
CALL UNIT_PTR (PUNIT,1,PSTREAM,CODE);
```

PUT_UPARM

```
CALL PUT_UPARM (PUNIT,4,13.9,CODE); OR CALL PUT_PARM (PPARM,4,13.9,CODE);
```

PUT_UPARMS

```
VALUEV(1)=2.3; VALUEV(2)=7.5; VALUEV(3)=1.6; VALUEV(4)=13.9;
```

```
CALL PUT_UPARMS (PUNIT,VALUEV,CODE); OR CALL PUT_PARMS (PPARM,VALUEV,CODE);
```

GET_UPARM

```
CALL GET_UPARM (PUNIT,2,VALUE,VTYPE,CODE); /* VALUE=7.5,VTYPE=3 */
```

```
OR
```

```
CALL GET_PARM (PPARM,2,VALUE,VTYPE,CODE);
```

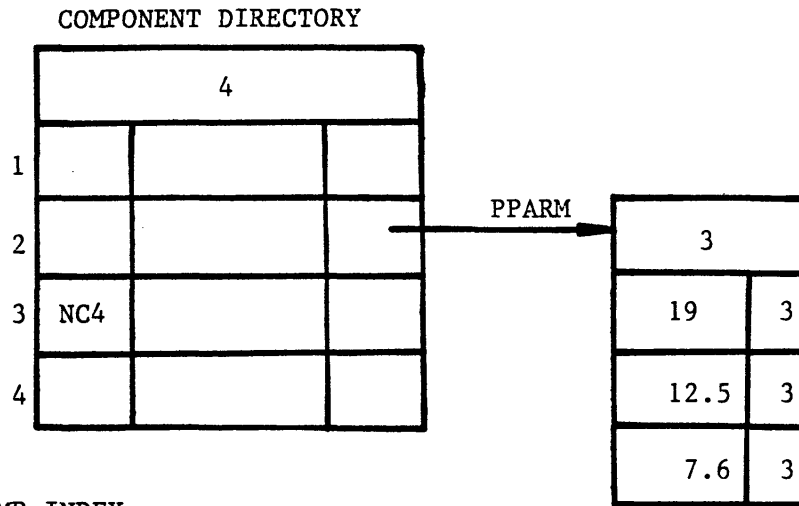
GET_UPARMS

```
CALL GET_UPARMS(PUNIT,VALUEV,VTV,CODE); OR CALL GET_PARMS (PUNIT,VALUEV,  
VTV,CODE);
```

```
/* VALUEV(1)=2.3,VALUEV(2)=7.5,VALUEV(3)=1.6,VALUEV(4)=13.9 */
```

```
/* VTV(1),VTV(2),VTV(3),VTV(4)=3 */
```

FIGURE C.1 EXAMPLES OF THE USE OF SERVICE ROUTINES
RELATED TO UNIT DATA STRUCTURES



```

GET_COMP_INDEX
CALL GET_COMP_INDEX ("NC4",COMP_INDEX,CODE);

/* COMP_INDEX=3 */

COMP_PTR
CALL COMP_PTR (2,PPARM,CODE);

PUT_CPARAM
CALL PUT_CPARAM (2,1,19,CODE);

PUT_CPARMS
VALUEV(1)=19; VALUEV(2)=12.5; VALUEV(3)=7.6;

CALL PUT_CPARMS (2,VALUEV,CODE);

GET_CPARAM
CALL GET_CPARAM(2,3,VALUE,VTYPE,CODE);

/* VALUE=7.6,VTYPE=3 */

GET_CPARMS
CALL GET_CPARMS(2,VALUEV,VTV,CODE);

/* VALUEV(1)=19,VALUEV(2)=12.5,VALUEV(3)=7.6 */

/* VTV(1),VTV(2),VTV(3)=3 */

```

FIGURE C.2 EXAMPLES OF THE USE OF SERVICE ROUTINES
RELATED TO COMPONENT DATA STRUCTURES

COMP_PTR

Given the component index, it returns the pointer to the component parameters' structure.

Usage

```
DCL COMP_PTR ENTRY (FIXED BIN, PTR, BIT (1));
```

```
CALL COMP_PTR (COMP_INDEX, PPARAM, CODE);
```

where:

COMP_INDEX is the component index (input).

PPARM is the pointer to the parameters' structure (output).

CODE is the error code (output)

Notes

CODE is turned "on" and P is null for the following cases:

- a. COMP_INDEX is negative or more than the maximum number of components (MAX_NC).
- b. The component does not exist.

Example

See Figure C.2.

FUNC_PTR

Given the function pointer, it returns the pointer to the function parameters structure.

Usage

```
DCL FUNC_PTR ENTRY (PTR, PTR, BIT(1));
```

```
CALL FUNC_PTR (PFUNC, PPARAM, CODE);
```

where:

PFUNC is the pointer to the function structure (input).

PPARM is the pointer to the parameters' structure (output).

CODE is the error code (output).

Notes

CODE is turned "on" and P is null when PFUNC is null.

Example

See Figure C.3.

STRM_PTR

Given the stream pointer, it returns the pointer to the parameters' structure of a given phase.

USAGE

```
DCL STRM_PTR ENTRY (PTR, FIXED BIN, PTR, BIT (1));
```

where:

PSTREAM is the pointer to the stream structure (input).

PHASE_NO is the phase number (input).

PPARM is the pointer to the parameters' structure (output).

CODE is the error code (output).

NOTES

CODE is turned "on" and P is null for the following cases:

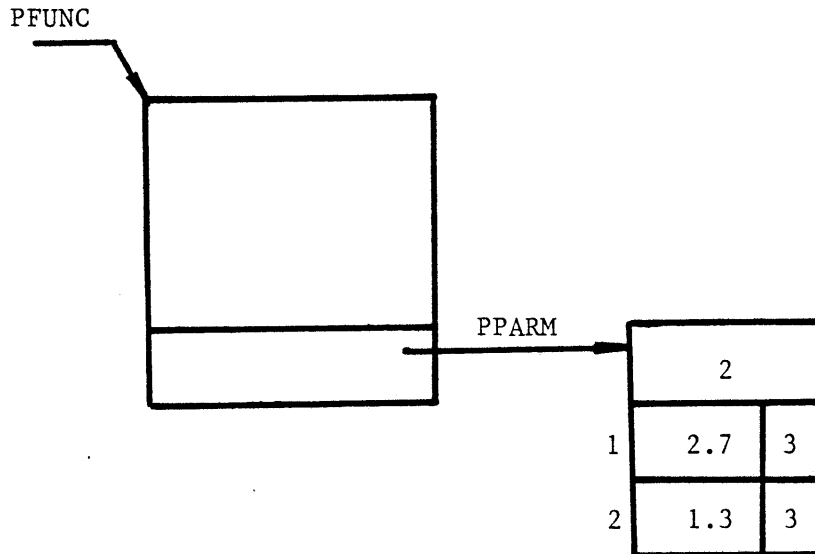
- a. PSTREAM is null.
- b. PHASE_NO is negative or greater than the number of phases of the stream.

Example

See Figure C.4

FLOW_PTR

It returns the pointer to the flow parameters structure, of a given component in a given phase of a stream.



```

FUNC_PTR
  CALL FUNC_PTR (PFUNC,PPARM,CODE);

PUT_FNPARM
  CALL PUT_FNPARM (PFUNC,2,1.3,CODE);

PUT_FNPARMS
  VALUEV(1)=2.7; VALUEV(2)=1.3;

  CALL PUT_FNPARMS (PFUNC,VALUEV,CODE);

GET_FNPARM
  CALL GET_FNPARM (PFUNC,1,VALUE,VTYPE,CODE);

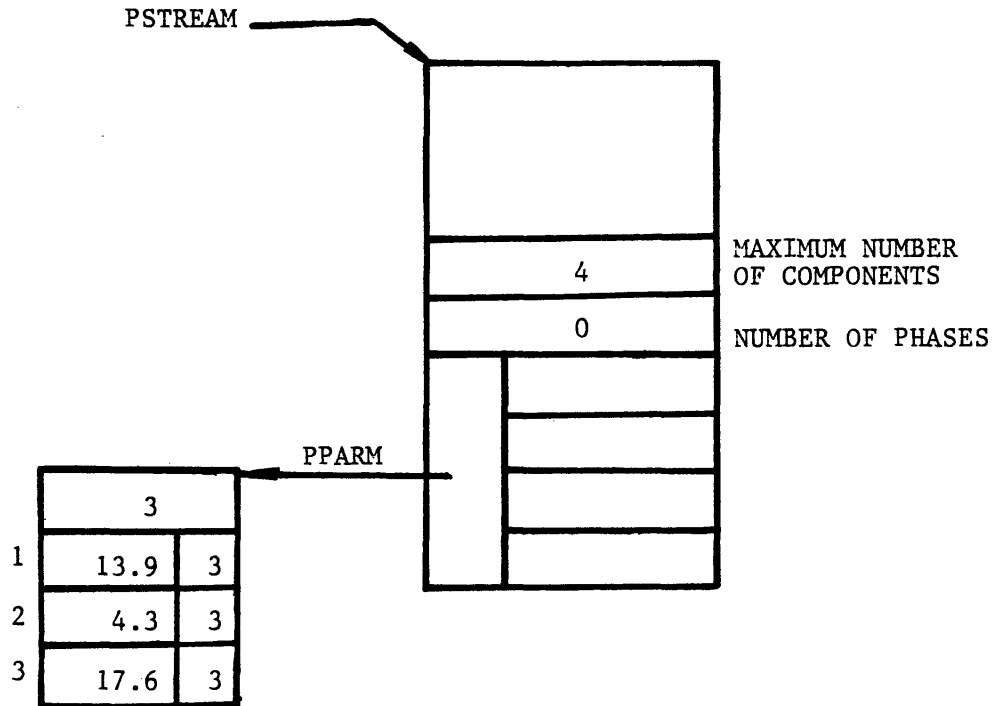
  /* VALUE=2.7 , VTYPE=3 */

GET_FNPARMS
  CALL GET_FNPARMS (PFUNC,VALUEV,VTV,CODE);

  /* VALUEV(1)=2.7, VALUEV(2)=1.3, VTV(1),VTV(2)=3 */

```

FIGURE C.3 EXAMPLES OF THE USE OF SERVICE ROUTINES
RELATED TO FUNCTION DATA STRUCTURES

STRM_PTR

```
CALL STRM_PTR (PSTREAM,0,PPARM,CODE);
```

PUT_SPARM

```
CALL PUT_SPARM (PSTREAM,0,2,4.3,CODE);
```

PUT_SPARMS

```
VALUEV(1)=13.9; VALUEV(2)=4.3; VALUEV(3)=17.6;
```

```
CALL PUT_SPARMS (PSTREAM,0,VALUEV,CODE);
```

GET_SPARM

```
CALL GET_SPARM (PSTREAM,0,1,VALUE,VTYPE,CODE);
```

```
/* VALUE=13.9, VTYPE=3 */
```

GET_SPARMS

```
CALL GET_SPARMS (PSTREAM,0,VALUEV,VTV,CODE);
```

```
/* VALUEV(1)=13.9, VALUEV(2)=4.3, VALUEV(3)=17.6 */
```

```
/* VTV(1), VTV(2), VTV(3) = 3 */
```

FIGURE C.4 EXAMPLES OF THE USE OF SERVICE ROUTINES
RELATED TO STREAM DATA STRUCTURES

USAGE

```
DCL    FLOW_PTR ENTRY (PTR, FIXED BIN, FIXED BIN, PTR, BIT (1));
CALL   FLOW_PTR (PSTREAM, PHASE_NO COMP_INDEX, PPARAM, CODE);
```

where:

```
PSTREAM    is the pointer to the stream structure (input).
PHASE_NO   is the phase number (input).
COMP_INDEX is the component index (input).
PPARAM     is the pointer to the flow parameters' structure (output).
CODE       is the error code (output).
```

NOTES

CODE is turned "on" and P is null for the following cases:

- a. PSTREAM is null.
- b. PHASE_NO is invalid, negative or greater than the number of phases of the stream.
- c. COMP_INDEX is invalid or the component does not exist or does not flow in the stream.

Example

See Figure C.5

C.2 Comparison Service RoutinesSAME_COMPS

Determines if the components in one stream are those that are in another stream.

Usage

```
DCL    SAME_COMPS ENTRY (PTR, PTR, BIT(1));
CALL   SAME_COMPS (PSTREAM1, PSTREAM2, CODE);
```

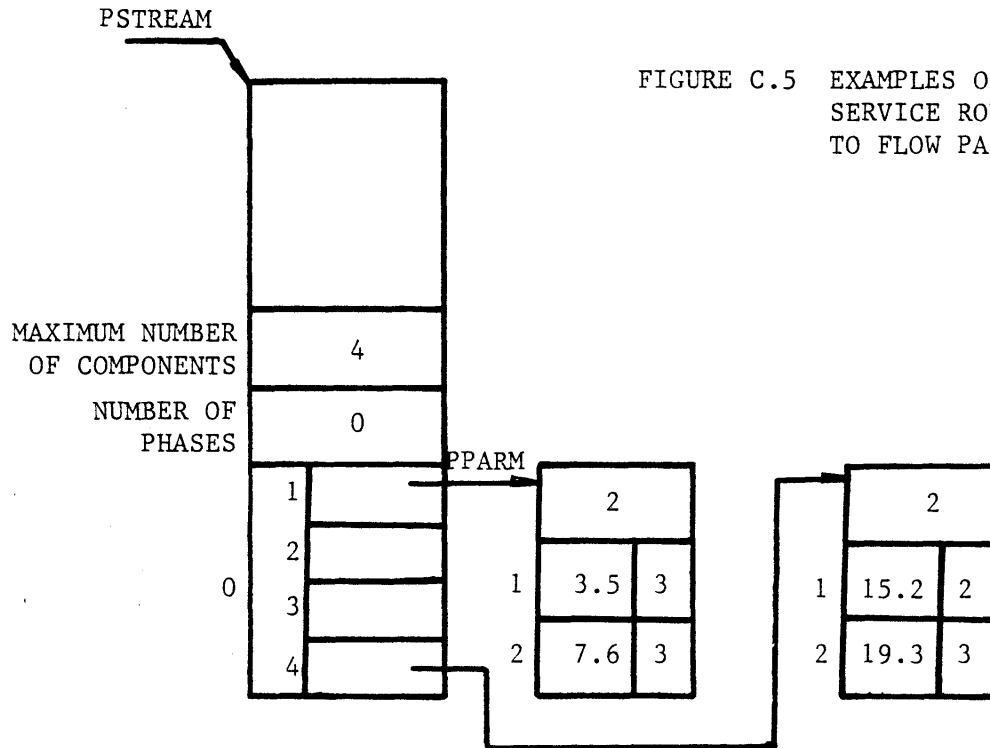


FIGURE C.5 EXAMPLES OF THE USE OF SERVICE ROUTINES RELATED TO FLOW PARAMETERS

FLOW_PTR

CALL FLOW_PTR (PSTREAM,0,1,PPARM,CODE);

PUT_FPARM

CALL PUT_FPARM (PSTREAM,0,1,1,3.5,CODE);

PUT_FPARMS

VALUEV(1)=3.5; VALUEV(2)=7.6;

CALL PUT_FPARMS (PSTREAM,0,1,VALUEV,CODE);

GET_FPARM

CALL GET_FPARM (PSTREAM,0,2,1,VALUE,VTYPE,CODE);

/* VALUE=15.2,VTYPE=2 */

GET_FPARMS

CALL GET_FPARMS(PSTREAM,0,1,VALUEV,VTV,CODE);

/* VALUEV(1)=3.5,VALUEV(2)=7.6,VTV(1),VTV(2)=3 */

PUT_FPARMACS

VALUEV(1)=7.6; VALUEV(4)=19.3;

CALL PUT_FPARMACS (PSTREAM,0,2,VALUEV,CODE);

GET_FPARMACS

CALL GET_FPARMACS (PSTREAM,0,2,VALUEV,VTV,CODE);

/* VALUEV(1)=7.6,VALUEV(4)=19.3,VTV(1),VTV(4)=3,VTV(2),VTV(3)=-1 */

where:

PSTREAM1 is the pointer to the 1st stream's structure (input).

PSTREAM2 is the pointer to the 2nd stream's structure (input).

CODE is the error code (output).

Notes

CODE is turned "on" for the following cases:

- a. PSTREAM1 or PSTREAM2 is null.
- b. The same components do not flow in both streams.

CHECK_COMPS

Determines if the components present in a stream are those whose names are specified.

Usage

```
DCL CHECK_COMPS ENTRY (PTR, DIM(*)CHAR(16),BIT(1));
```

```
CALL CHECK_COMPS (PSTREAM, COMP_NAMES, CODE);
```

where:

PSTREAM is the pointer to the streams' structure (input).

COMP_NAMES is the array of component names (input).

CODE is the error code (output).

Notes

CODE is turned "on" for the following cases:

- a. PSTREAM is null.
- b. Components in the stream are not exactly those specified in the array of component names.

C.3 Service Routines Retrieving or Storing the Values of Parameters

GET PARM, XGET PARM, IGET PARM

Each of these routines retrieves the value and value type of a given parameter. The only difference among the routines is the attribute of the returned value.

Usage

```
DCL GET_PARM ENTRY (PTR, FIXED BIN, FLOAT BIN(63), FIXED BIN, BIT(1));
DCL XGET_PARM ENTRY (PTR, FIXED BIN, FLOAT BIN(27), FIXED BIN,
BIT(1));
DCL IGET_PARM (PTR, FIXED BIN, FIXED BIN, FIXED BIN, BIT(1));
CALL GET_PARM (PPARM, PARM_NO, VALUE, VTYPE, CODE);
CALL XGET_PARM (PPARM, PARM_NO, XVALUE, VTYPE, CODE);
CALL IGET_PARM (PPARM, PARM_NO, IVALUE, VTYPE, CODE);
```

where:

PPARM is the pointer to the parameters' structure (input).
 PARM_NO is the parameter number (input).
 VALUE is the parameters' value in double precision (output).
 XVALUE is the parameter's value in single precision (output).
 IVALUE is the parameter's value as an integer number (output).
 VTYPE is the value type (output).
 CODE is the error code (output).

Notes

CODE is "on" for the following cases:

- a. PPARM is null.
- b. PARM_NO is invalid.

VTYPE has one of the following values:

- 0 Unspecified

- 1 Assumed
- 2 Specified
- 3 Calculated

Example

See Figure C.6

GET_PARMS, XGET_PARMS, IGET_PARMS

Each of these routines retrieves the values and value types of all parameters of a given parameters' structure. The only difference among the routines is the attribute of the returned values.

Usage

```
DCL   GET_PARMS ENTRY (PTR, DIM (*) FLOAT BIN (63), DIM (*) FIXED
      BIN, BIT (1));

DCL   XGET_PARMS ENTRY (PTR, DIM (*) FLOAT BIN (27), DIM (*) FIXED
      BIN, BIT (1));

DCL   IGET_PARMS ENTRY (PTR, DIM (*) FIXED BIN, DIM (*) FIXED BIN,
      BIT (1));

CALL  GET_PARMS (PPARM, VALUEV, VTV, CODE);

CALL  XGET_PARMS (PPARM, XVALUEV, VTV, CODE);

CALL  IGET_PARMS (PPARM, IVALUEV, VTV, CODE);
```

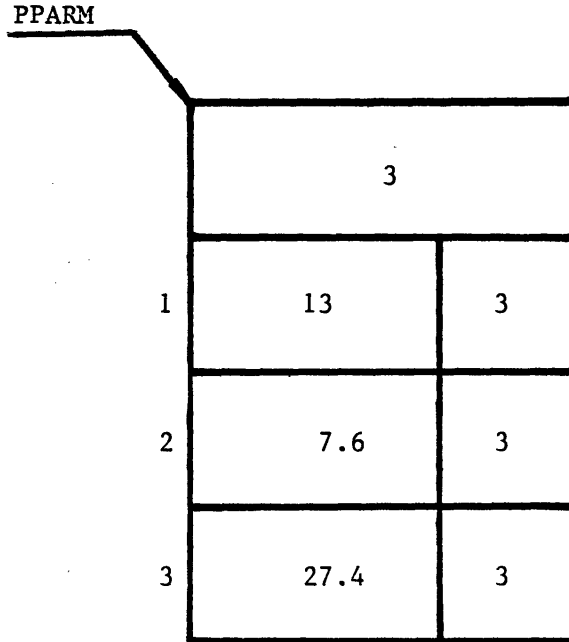
where:

PPARM is the pointer to the parameters' structure (input).

VALUEV is the vector containing the parameters' values in double precision (output).

XVALUEV is the vector containing the parameters' values in single precision (output).

IVALUEV is the vector containing the parameters' values as integer numbers (output).

PUT_PARM

```
CALL PUT_PARM (PPARM,2,7.6,CODE);
```

PUT_PARMS

```
VALUEV(1)=13; VALUEV(2)=7.6; VALUEV(3)=27.4;
```

```
CALL PUT_PARMS (PPARM,VALUEV,CODE);
```

GET_PARM

```
CALL GET_PARM (PPARM,1,VALUE,VTYPE,CODE);
```

```
/* VALUE=13,VTYPE=3 */
```

GET_PARMS

```
CALL GET_PARMS (PPARM,VALUEV,VTV,CODE);
```

```
/* VALUEV(1)=13,VALUEV(2)=7.6,VALUEV(3)=27.4 */
```

```
/* VTV(1),VTV(2),VTV(3)=3 */
```

FIGURE C.6 EXAMPLES OF THE USE OF SERVICE ROUTINES
DIRECTLY RETRIEVING OR STORING THE VALUES OF PARAMETERS

VTV is the vector containing the value types(output).

CODE is the error code(output).

Notes

CODE is "on" for the following cases:

- a. PPARAM is null.
- b. The size of VALUEV, XVALUEV, IVALUEV, OR VTV is not equal to the number of the parameters.

The Ith entry of VTV indicates the value type of the Ith parameter.

Each entry of VTV may be one of the following:

- | | |
|---|-------------|
| 0 | UNSPECIFIED |
| 1 | ASSUMED |
| 2 | SPECIFIED |
| 3 | CALCULATED |

Example

See Figure C.6

PUT PARM. XPUT PARM. IPUT PARM

Each of these routines stores the value of a given parameter. The only difference among the routines is the attribute of the given value.

Usage

```
DCL    PUT_PARM ENTRY(PTR, FIXED BIN,  FLOAT BIN (63), BIT (1));
DCL    XPUT_PARM ENTRY (PTR, FIXED BIN,  FLOAT BIN (27), BIT (1));
DCL    IPUT_PARM ENTRY (PTR, FIXED BIN,  FIXED BIN, BIT(1));
CALL   PUT_PARM (PPARM, PARM_NO, VALUE, CODE);
CALL   XPUT_PARM (PPARM, PARM_NO, XVALUE, CODE);
CALL   IPUT_PARM (PPARM, PARM_NO, IVALUE, CODE);
```

where:

PPARM is the pointer to the parameters' structure (input).

PARAM_NO is the parameter number (input).

VALUE is the parameter's value in double precision (input).

XVALUE is the parameter's value in single precision (input).

IVALUE is the parameter's value as an integer number (input).

CODE is the error code (output).

Notes

CODE is "on" for the following cases:

- a. PPARM is null
- b. PARAM_NO is invalid (less than one or greater than number of parameters).

The value type of 3 (calculated) will be assigned to the parameter.

Example

Assign the result of some calculations, RESULT, to the 3rd parameter of a unit. The pointer to the parameters' structure, PPARM, has been already found.

```
CALL PUT_PARAM (PPARM, 3, RESULT, CODE);
```

See also Figure C.6.

PUT PARMS, XPUT PARMS, IPUT PARMS

Each of these routines stores the values of all parameters of a given set. The only difference among the routines is the attribute of the given values.

Usage

```
DCL PUT_PARMS ENTRY (PTR, DIM(*) FLOAT BIN(63), BIT(1));
```

```
DCL XPUT_PARMS ENTRY (PTR, DIM(*) FLOAT BIN(27), BIT(1));
```

```

DCL   IPUT_PARMS ENTRY (PTR, DIM(*) FIXED BIN, BIT(1));
CALL  PUT_PARMS (PPARM, VALUEV, CODE);
CALL  XPUT-PARMS (PPARM, XVALUEV, CODE);
CALL  IPUT_PARMS (PPARM, IVALUEV, CODE);

```

where:

PPARM is the pointer to the parameters' structure (input).

VALUEV is the vector containing the parameter values in double precision (input).

XVALUEV is the vector containing the parameter values in single precision (input).

IVALUEV is the vector containing the parameter values as integer numbers (input).

CODE is the error code (output).

Notes

CODE is "On" for the following cases:

- a. PPARM is null.
- b. The size of vector VALUEV, XVALUEV, or IVALUEV is not equal to the number of parameters.

The value type of 3 (calculated) will be assigned to all parameters.

Example

Suppose there is a parameters' structure having 10 parameters. All the parameters have been calculated and the result is in vector XVALUEV. Store the result in the process network.

```

DCL  XVALUE(10) FLOAT BIN;
DCL  XPUT_PARMS ENTRY (PTR, DIM(*) FLOAT BIN, BIT(1));

```

.
.
.

```
CALL XPUT_PARM (PPARM, XVALUEV, CODE);
```

See also Figure C.6.

GET UPARM. XGET UPARM. IGET UPARM

Each of these routines retrieves the value and value type of a given unit parameter. The only difference among the routines is the attribute of the returned value.

Usage

```
DCL   GET_UPARM_ENTRY (PTR, FIXED BIN, FLOAT BIN(63), FIXED BIN,
        BIT(1));

CALL   GET_UPARM (PUNIT, PARM_NO, VALUE, VTYPE, CODE);

DCL   XGET_UPARM_ENTRY (PTR, FIXED BIN, FLOAT BIN (27), FIXED BIN,
        BIT(1));

CALL   XGET_UPARM (PUNIT, PARM_NO, XVALUE, VTYPE, CODE);

DCL   IGET_PARM_ENTRY (PTR, FIXED BIN, FIXED BIN, FIXED BIN,
        BIT(1));

CALL   IGET_UPARM (PUNIT, PARM_NO, IVALUE, VTYPE, CODE);
```

where:

PUNIT is the pointer to the unit structure (input).

Other symbols are as defined for the GET_PARM routine.

Notes

The same as those given for the GET_PARM routine.

Example

See Figure C.1.

GET UPARMS. XGET UPARMS. IGET UPARMS

Each of these routines retrieves the values and value types of all

parameters of a given unit. The only difference among the routines is the attribute of the returned values.

Usage

```
DCL  GET_UPARMS ENTRY(PTR, DIM(*) FLOAT BIN(63), DIM(*) FIXED BIN, BIT(1));
DCL  GET_UPARMS ENTRY(PTR, DIM(*) FLOAT BIN(27), DIM(*) FIXED BIN, BIT(1));
DCL  IGET_UPARMS ENTRY(PTR, DIM(*) FIXED BIN, DIM(*) FIXED BIN, BIT(1));
CALL  GET_UPARMS (PUNIT, VALUEV ,VTV, CODE);
CALL  XGET_UPARMS (PUNIT, XVALUEV ,VTV, CODE);
CALL  IGET_UPARMS (PUNIT, IVALUEV ,VTV, CODE);
```

where:

PUNIT is the pointer to the unit structure (input)

Other symbols are as defined for GET_PARMS routine.

Notes

See those given for GET_PARMS routine.

Example

See Figure C.1.

PUT UPARM, XPUT PARM, IPUT PARM

Each of these routines stores the value of a given unit parameter.

The only difference among the routines is the attribute of the given value.

Usage

```
DCL  PUT_UPARM  ENTRY(PTR, FIXED BIN,  FLOAT BIN(63), BIT(1));
DCL  XPUT_UPARM ENTRY (PTR, FIXED BIN,  FLOAT BIN(27),  BIT(1));
DCL  IPUT_UPARM ENTRY(PTR, FIXED BIN,  FIXED BIN,    BIT(1));
CALL  PUT_UPARM  (PUNIT, PARM_NO,      VALUE,      CODE);
CALL  XPUT_UPARM (PUNIT, PARM_NO,      XVALUE,     CODE);
CALL  IPUT_UPARM (PUNIT, PARM_NO,      IVALUE,     CODE);
```

where:

PUNIT is the pointer to the unit structure (input).

Other symbols are as defined for PUT_PARM routine.

Notes

See those given for PUT_PARM routine.

Example

See figure C.1.

PUT UPARMS, XPUT UPARMS, IPUT UPARMS

Each of these routines stores the values of all parameters of a given unit.

The only difference among the routines is the attribute of the given values.

Usage

```

DCL   PUT_UPARMS   ENTRY(PTR,   DIM(*) FLOAT BIN(63), BIT(1));
DCL   XPUT_UPARMS ENTRY (PTR,   DIM(*) FLOAT BIN(27), BIT(1));
DCL   IPUT_UPARMS ENTRY(PTR,   DIM(*) FIXED BIN,      BIT(1));
CALL  PUT_UPARMS   (PUNIT,     VALUEV      ,CODE);
CALL  XPUT_UPARMS (PUNIT,     XVALUEV     ,CODE);
CALL  IPUT_UPARMS (PUNIT,     IVALUEV     ,CODE);

```

where:

PUNIT is the pointer to the unit structure (input).

Other symbols are as defined for PUT_PARMS routine.

Notes

See those given for PUT_PARMS routine.

Example

See Figure C.1.

GET CPARM. XGET CPARM. IGET CPARM

Each of these routines retrieves the value and value type of a given component parameter. The only difference among the routines is the attribute of the returned value.

Usage

```
DCL GET_CPARM ENTRY (FIXED BIN, FIXED BIN, FLOAT BIN (63), FIXED BIN, BIT(1));
DCL XGET_CPARM ENTRY (FIXED BIN, FIXED BIN, FLOAT BIN, FIXED BIN, BIT(1));
DCL IGET_CPARM ENTRY (FIXED BIN, FIXED BIN, FIXED BIN, FIXED BIN, BIT(1));
CALL GET_CPARM (COMP_INDEX, PARM_NO, VALUE, VTYPE, CODE);
CALL XGET_CPARM (COMP_INDEX, PARM_NO, XVALUE, VTYPE, CODE);
CALL IGET_CPARM (COMP_INDEX, PARM_NO, IVALUE, VTYPE, CODE);
```

where:

COMP_INDEX is the component index (input). Other symbols are as defined for GET_PARM routine.

Notes

See those given for GET_PARM routine.

Example

See Figure C.1

GET CPARMS. XGET CPARMS. IGET CPARMS

Each of these routines retrieves the values and value types of all parameters of a given component. The only difference among the routines is the attribute of the returned values.

```
DCL GET_CPARMS ENTRY(FIXED BIN, DIM(*) FLOAT BIN (63),
DIM(*) FIXED BIN, BIT(1));
DCL XGET_CPARMS ENTRY (FIXED BIN, DIM(*) FLOAT BIN (27),
DIM(*) FIXED BIN, BIT(1));
```

```
DCL IGET_CPARMS ENTRY (FIXED BIN, DIM(*) FIXED BIN,
                        DIM(*) FIXED BIN, BIT(1));
```

```
CALL GET_CPARMS (COMP_INDEX, VALUEV, VTV, CODE);
```

```
CALL XGET_CPARMS (COMP_INDEX, XVALUEV, VTV, CODE);
```

```
CALL IGET_CPARMS (COMP_INDEX, IVALUEV, VTV, CODE);
```

where:

COMP_INDEX is the component index (input). Other symbols are as defined for the GET_PARM routine.

Notes

See those given for the GET_PARM routine.

Example

See Figure C.2

PUT_CPARM. XPUT_CPARM. IPUT_CPARM

Each of these routines stores the value of a given component parameter. The only difference among the routines is the attribute of the given value.

Usage

```
DCL PUT_CPARM ENTRY (FIXED BIN, FIXED BIN, FLOAT BIN (63), BIT(1));
```

```
DCL XPUT_CPARM ENTRY (FIXED BIN, FIXED BIN, FLOAT BIN (27), BIT(1));
```

```
DCL IPUT_CPARM ENTRY (FIXED BIN, FIXED BIN, FIXED BIN, BIT (1));
```

```
CALL PUT_CPARM (COMP_INDEX, PARM_NO, VALUE, CODE);
```

```
CALL XPUT_CPARM (COMP_INDEX, PARM_NO, XVALUE, CODE);
```

```
CALL IPUT_CPARM (COMP_INDEX, PARM_NO, IVALUE, CODE);
```

where:

COMP_INDEX is the component index (input). Other symbols are as defined for the PUT_PARM routine.

Notes

See those given for PUT_PARM routine.

Example

See Figure C.2

PUT_CPARMS, XPUT_CPARMS, IPUT_CPARMS

Each of these routines stores the values of all parameters of a given component. The only difference among the routines is the attribute of the given values.

Usage

```
DCL    PUT_CPARMS ENTRY (FIXED BIN, DIM (*) FLOAT BIN (63), BIT(1));
DCL    XPUT_CPARMS ENTRY (FIXED BIN, DIM (*) FLOAT BIN (27), BIT(1));
DCL    IPUT_CPARMS ENTRY (FIXED BIN, DIM (*) FIXED BIN, BIT (1));
CALL   PUT_CPARMS (COMP_INDEX, VALUE, CODE);
CALL   XPUT_CPARMS (COMP_INDEX, XVALUEV, CODE);
CALL   IPUT_CPARMS (COMP_INDEX, IVALUE, CODE);
```

where:

COMP_INDEX is the component index (input). Other symbols are as defined for PUT_PARMS routine.

Notes

See those given for PUT_PARMS routine.

Example

See Figure C.2

GET_FNPARM.XGET_FNPARM.IGET_FNPARM

Each of these routines retrieves the value and value type of a given function parameter. The only difference among the routines is the attribute of the returned value.

Usage

```
DCL GET_FNPARM ENTRY (PTR, FIXED BIN, FLOAT BIN (63), FIXED BIN, BIT(1));
DCL XGET_FNPARM ENTRY (PTR, FIXED BIN, FLOAT BIN (27), FIXED BIN, BIT(1));
DCL IGET_FNPARM ENTRY (PTR, FIXED BIN, FIXED BIN, FIXED BIN, BIT(1));
CALL GET_FNPARM (PFUNC, PARM_NO, VALUE, VTYPE, CODE);
CALL XGET_FNPARM (PFUNC, PARM_NO, XVALUE, VTYPE, CODE);
CALL IGET_FNPARM (PFUNC, PARM_NO, IVALUE, VTYPE, CODE);
```

where:

PFUNC is the pointer to the function structure (input). Other symbols are as defined for the GET_PARM routine.

Notes

See those given for the GET_PARM routine.

Example

See Figure C.3

GET_FNPARMS.XGET_FNPARMS.IGET_FNPARMS

Each of these routines retrieves the values and value types of all parameters of a given function. The only difference among the routines is the attribute of the returned values.

Usage

```
DCL GET_FNPARMS ENTRY (PTR, DIM (*) FLOAT BIN (63),
DIM (*) FIXED BIN, BIT(1));
```

```

DCL   XGET_FNPARMS  ENTRY (PTR,          DIM (*) FLOAT BIN (27),
                                     DIM (*) FIXED BIN,  BIT(1));
DCL   IGET_FNPARMS  ENTRY (PTR,          DIM (*) FIXED BIN,
                                     DIM (*) FIXED BIN,  BIT(1));

CALL  GET_FNPARMS   (PFUNC,          VALUEV,  VTV,  CODE);
CALL  XGET_FNPARMS (PFUNC ,          XVALUEV, VTV,  CODE);
CALL  IGET_FNPARMS (PFUNC,          IVALUEV, VTV,  CODE);

```

where:

PFUNC is the pointer to the function's structure (input). Other symbols are as defined for the GET_PARMs routine.

Notes

The same as those given for GET_PARMs routine.

Example

See Figure C.3

PUT_FNPARM. XPUT_FNPARM. IPUT_FNPARM

Each of these routines stores the value of a given function parameter. The only difference among these routines is the attribute of the given value.

Usage

```

DCL  PUT_FNPARM  ENTRY(PTR, FIXED BIN, FLOAT BIN (63), BIT(1));
DCL  XPUT_FNPARM ENTRY(PTR, FIXED BIN, FLOAT BIN (27), BIT(1));
DCL  IPUT_FNPARM ENTRY(PTR, FIXED BIN, FIXED BIN,      BIT(1));
CALL PUT_FNPARM  (PFUNC, PARM_NO, VALUE, CODE);
CALL XPUT_FNPARM (PFUNC, PARM_NO, XVALUE, CODE);
CALL IPUT_FNPARM (PFUNC, PARM_NO, IVALUE, CODE):

```

where:

PFUNC is the pointer to the function's structure (input). Other symbols are as defined for the PUT_PARM routine.

Notes

The same as those given for the PUT_PARM routine.

Example

See Figure C.3

PUT FNPARMS, XPUT FNPARMS, IPUT FNPARMS

Each of these routines stores the values of all parameters of a given function. The only difference among the routines is the attribute of the given values.

Usage

```

DCL   PUT_FNPARMS   ENTRY (PTR, DIM (*) FLOAT BIN (63), BIT(1));
DCL   XPUT_FNPARMS ENTRY (PTR, DIM (*) FLOAT BIN (27), BIT(1));
DCL   IPUT_FNPARMS ENTRY (PTR, DIM (*) FIXED BIN,      BIT (1);

CALL  PUT_FNPARMS   (PFUNC,      VALUEV, CODE);
CALL  XPUT_FNPARMS (PFUNC,      XVALUEV, CODE);
CALL  IPUT_FNPARMS (PFUNC,      IVALUEV, CODE);

```

where:

PFUNC is the pointer to the function's structure (input). Other symbols are as defined for the PUT_PARMS routine.

Notes

The same as those given for the PUT_PARMS routine.

Example

See Figure C.3

GET SPARM, XGET SPARM, IGET SPARM

Each of these routines retrieves the value and value type of a phase (stream) parameter of a given phase of a stream. The only difference among the routines is the attribute of the returned value.

Usage

```
DCL GET_SPARM ENTRY (PTR, FIXED BIN, FIXED BIN, FLOAT BIN (63), BIT(1));
```

```
DCL XGET_SPARM ENTRY (PTR, FIXED BIN, FIXED BIN, FLOAT BIN (27), BIT(1));
```

```
DCL IGET_SPARM ENTRY (PTR, FIXED BIN, FIXED BIN, FIXED BIN, BIT(1));
```

```
CALL GET_SPARM (PSTREAM, PHASE_NO, PARM_NO, VALUE, VTYPE, CODE);
```

```
CALL XGET_SPARM (PSTREAM, PHASE_NO, PARM_NO, XVALUE, VTYPE, CODE);
```

```
CALL IGET_SPARM (PSTREAM, PHASE_NO, PARM_NO, IVALUE, VTYPE, CODE);
```

where:

PSTREAM is the pointer to the stream structure (input).

PHASE_NO is the phase number (input).

Other symbols are as defined for the GET_PARM routine.

Notes

The same as those given for GET_PARM routine.

Example

See Figure C.4

GET SPARMS, XGET SPARMS, IGET SPARMS

Each of these routines retrieves the values and value types of all phase (stream) parameters of a given phase of a stream. The only difference among the routines is the attribute of the given values.

Usage

```
DCL PUT_SPARMS ENTRY (PTR, FIXED BIN, DIM (*) FLOAT BIN (63), BIT(1));
```

```
DCL XPUT_SPARMS ENTRY (PTR, FIXED BIN, DIM (*) FLOAT BIN (27), BIT(1));
```

```
DCL IPUT_SPARMS ENTRY (PTR, FIXED BIN, DIM (*) FIXED BIN, BIT (1));
```

```
CALL PUT_SPARMS (PSTREAM, PHASE_NO, VALUEV, CODE);
```

```
CALL XPUT_SPARMS (PSTREAM, PHASE_NO, XVALUEV, CODE);
```

```
CALL IPUT_SPARMS (PSTREAM, PHASE_NO, IVALUEV, CODE);
```

where:

PSTREAM is the pointer to the stream structure (input).

PHASE_NO is the phase number (input).

Other symbols are as defined for PUT_PARMS routine.

Notes

See those given for PUT_PARMS routine.

Example

See Figure C.4

PUT SPARM. XPUT SPARM. IPUT SPARM

Each of these routines stores the value of a phase (stream) parameter of a given phase of a stream. The only difference among the routines is the attribute of the given value.

Usage

```
DCL PUT_SPARM ENTRY (PTR, FIXED BIN, FIXED BIN, FLOAT BIN (63), BIT(1));
```

```
DCL XPUT_SPARM ENTRY (PTR, FIXED BIN, FIXED BIN, FLOAT BIN (27), BIT(1));
```

```
DCL IPUT_SPARM ENTRY (PTR, FIXED BIN, FIXED BIN, FIXED BIN, BIT(1));
```



```
CALL PUT_SPARM (PSTREAM, PHASE_NO, PARM_NO, VALUE, CODE);
CALL XPUT_SPARM (PSTREAM, PHASE_NO, PARM_NO, XVALUE, CODE);
CALL IPUT_SPARM (PSTREAM, PHASE_NO, PARM_NO, IVALUE, CODE);
```

where:

PSTREAM is the pointer to the stream structure (input).

PHASE_NO is the phase number (input).

Other symbols are as defined for the PUT_PARM routine.

Notes

The same as those given for PUT_PARM routine.

Example

See Figure C.4

PUT SPARMS, XPUT SPARMS, IPUT SPARMS

Each of these routines stores the values of all phase (stream) parameters of a given phase of a stream. The only difference among the routines is the attribute of the given values.

Usage

```
DCL PUT_SPARMS ENTRY (PTR, FIXED BIN, DIM(*) FLOAT BIN (63), BIT(1));
DCL XPUT_SPARMS ENTRY (PTR, FIXED BIN, DIM(*) FLOAT BIN (27), BIT(1));
DCL IPUT_SPARMS ENTRY (PTR, FIXED BIN, DIM(*) FIXED BIN , BIT(1));

CALL PUT_SPARMS (PSTREAM, PHASE_NO, VALUEV, CODE);
CALL XPUT_SPARMS (PSTREAM, PHASE_NO, XVALUEV, CODE);
CALL IPUT_SPARMS (PSTREAM, PHASE_NO, IVALUEV, CODE);
```

where:

PSTREAM is the pointer to the stream structure (input).

PHASE_NO is the phase number (input).

Other symbols are as defined for PUT_PARM routine.

Notes

See those given for PUT_PARM routine.

Example

See Figure C.4.

GET_FPARM, XGET_FPARM, IGET_FPARM

Each of these routines retrieves the value and value type of a flow parameter of a component in a given phase of a stream. The only difference among the routines is the attribute of the returned value.

Usage

```
DCL GET_FPARM ENTRY (PTR, FIXED BIN, FIXED BIN, FIXED BIN,
                    FLOAT BIN (63), FIXED BIN, BIT(1));

DCL XGET_FPARM ENTRY (PTR, FIXED BIN, FIXED BIN, FIXED BIN,
                    FLOAT BIN (27), FIXED BIN, BIT(1));

DCL IGET_FPARM ENTRY (PTR, FIXED BIN, FIXED BIN, FIXED BIN,
                    FIXED BIN, FIXED BIN, BIT(1));
```

```
CALL GET_FPARM (PSTREAM, PHASE_NO, COMP_INDEX, PARM_NO, VALUE, VTYPE, CODE);
```

```
CALL XGET_FPARM (PSTREAM, PHASE_NO, COMP_INDEX, PARM_NO, XVALUE, VTYPE, CODE);
```

```
CALL IGET_FPARM (PSTREAM, PHASE_NO, COMP_INDEX, PARM_NO, IVALUE, VTYPE, CODE);
```

where:

PSTREAM is the pointer to the stream structure (input).

PHASE_NO is the phase number (input).

COMP_INDEX is the component index (input).

Other symbols are as defined for GET_PARM routine.

Notes

See those given for GET_PARM routine.

Example

See Figure C.5.

GET_FPARMS. XGET_FPARMS. IGET_FPARMS

Each of these routines retrieves the values and value types of all flow parameters of a component in a given phase of a stream. The only difference among the routines is the attribute of the returned values.

Usage

```
DCL GET_FPARMS ENTRY (PTR, FIXED BIN, FIXED BIN, DIM (*) FLOAT BIN (63),
                    DIM (*) FIXED BIN, BIT(1));
```

```
DCL XGET_FPARMS ENTRY (PTR, FIXED BIN, FIXED BIN, DIM (*) FLOAT BIN (27),
                    DIM (*) FIXED BIN, BIT(1));
```

```
DCL IGET_FPARMS ENTRY (PTR, FIXED BIN, FIXED BIN, DIM (*) FIXED BIN,
                    DIM(*) FIXED BIN, BIT(1));
```

```
CALL GET_FPARMS (PSTREAM, PHASE_NO, COMP_INDEX, VALUEV, VTV, CODE);
```

```
CALL XGET_FPARMS (PSTREAM, PHASE_NO, COMP_INDEX, XVALUE, VTV, CODE);
```

```
CALL IGET_FPARMS (PSTREAM, PHASE_NO, COMP_INDEX, IVALUE, VTV, CODE);
```

Where:

PSTREAM is the pointer to the stream structure (input).

PHASE_NO is the phase number (input).

COMP_INDEX is the component index (input).

Other symbols are as defined for GET_PARDS routine.

Notes

See those given for GET_PARDS routine.

Example:

See Figure C.5

PUT FPARM. XPUT FPARM. IPUT FPARM

Each of these routines stores the value of a flow parameter of a component in a given phase of a stream. The only difference among the routines is the attribute of the given value.

Usage

```
DCL  PUT_FPARM ENTRY(PTR, FIXED BIN, FIXED BIN, FIXED BIN,
                    FLOAT BIN(63), BIT(1));

DCL  XPUT_FPARM ENTRY(PTR, FIXED BIN, FIXED BIN, FIXED BIN,
                    FLOAT BIN(27), BIT(1));

DCL  IPUT_FARM ENTRY(PTR, FIXED BIN, FIXED BIN, FIXED BIN,
                    FIXED BIN, BIT(1));

CALL PUT_FPARM (PSTREAM, PHASE_NO, COMP_INDEX, PARM_NO, VALUE, CODE);

CALL XPUT_FPARM (PSTREAM, PHASE_NO, COMP_INDEX, PARM_NO, XVALUE, CODE);
```

```
CALL IPUT_FPARM (PSTREAM,PHASE_NO,COMP_INDEX,PARM_NO,IVALUE,CODE);
```

where:

PSTREAM is the pointer to the stream structure (input).

PHASE_NO is the phase number (input).

COMP_INDEX is the component Index (input).

Other symbols are as defined for PUT_PARM routine.

Notes

See those given for PUT_PARM routine.

Example

See Figure C.5.

PUT FPARMS. XPUT FPARMS. IPUT FPARMS

Each of these routines stores the values of all flow parameters of a component in a given phase of a stream. The only difference among the routines is the attribute of the given values.

```
DCL PUT_FPARMS ENTRY (PTR, FIXED BIN, FIXED BIN, DIM (*)  
                      FLOAT BIN (63), BIT(1));
```

```
DCL XPUT_FPARMS ENTRY (PTR, FIXED BIN, FIXED BIN, DIM (*)  
                      FLOAT BIN, BIT(1));
```

```
DCL IPUT_FPARMS ENTRY (PTR, FIXED BIN, FIXED BIN, DIM (*)  
                      FIXED BIN, BIT(1));
```

```
CALL PUT_FPARMS (PSTREAM,PHASE_NO,COMP_INDEX,VALUEV,CODE);
```

```
CALL XPUT_FPARMS (PSTREAM,PHASE_NO,COMP_INDEX,XVALUE,CODE);
```

```
CALL IPUT_FPARMS (PSTREAM, PHASE_NO, COMP_INDEX, IVALUE, CODE);
```

Where:

PSTREAM is the pointer to the stream structure (input).

PHASE_NO is the phase number (input).

COMP_INDEX is the component index (input).

Other symbols are as defined for PUT_PARM routine.

Notes

See those given for PUT_PARM routines.

Example

See Figure C.5.

GET_FPARMACS. XGET_FPARMACS. IGET_FPARMACS

Each of these routines retrieves the values and value types of a given flow parameter of all components in a given phase of a stream. The only difference between the routines is the attribute of the returned values.

Usage

```
DCL GET_FPARMACS ENTRY (PTR, FIXED BIN, FIXED BIN, DIM(*) FLOAT BIN(63),
                        DIM(*) FIXED BIN, BIT(1));
```

```
DCL XGET_FPARMACS ENTRY (PTR, FIXED BIN, FIXED BIN, DIM(*) FLOAT BIN(27),
                        DIM(*) FIXED BIN, BIT(1));
```

```
DCL IGET_FPARMACS ENTRY (PTR, FIXED BIN, FIXED BIN, DIM(*) FIXED BIN,
                        DIM(*) FIXED BIN, BIT(1));
```

```
CALL GET_FPARMACS (PSTREAM, PHASE_NO, PARM_NO, VALUEV, VTV, CODE);
```

```
CALL XGET_FPARMACS (PSTREAM, PHASE_NO, PARM_NO, XVALUEV, VTV, CODE);
```

```
CALL IGET_FPARMACS (PSTREAM, PHASE_NO, PARM_NO, IVALUEV, VTV, CODE);
```

where:

PSTREAM is the pointer to the stream's structure (input).

PHASE_NO is the phase number (input).

PARM_NO is the flow parameter number (input).

VALUEV, XVALUEV, IVALUEV are vectors containing the values of the given flow parameter of all components in the given phase of the stream (output).

VALUEV is in double precision.

XVALUEV is in single precision.

IVALUEV is in integers.

VTV is the value type vector, which contains the value types of the given flow parameter of all components in the given phase of the stream (output).

CODE is the error code (output).

Notes

CODE is "on" for the following cases:

- a. PSTREAM is null.
- b. PHASE_NO is invalid (negative, or greater than the number of phases of the stream).
- c. PARM_NO is invalid (less than one, or greater than the number of flow parameters of the given phase of the stream).
- d. The dimension size of VALUEV, XVALUEV, IVALUEV, or VTV is not equal to the maximum number of components (MAX_NC).

VTV(I) contains the value type of the given flow parameter of the Ith component.

If $VTV(I)=-1$ indicates that the I th component does not flow in the stream.

$VTV(I) = 0$ indicates that the flow parameter of I th component is unspecified.

$VTV(I) = 1$ indicates that the flow parameter of I th component is assumed.

$VTV(I) = 2$ indicates that the flow parameter of I th component is specified.

$VTV(I) = 3$ indicates that the flow parameter of I th component is calculated.

$VALUEV(I)$, $XVALUEV(I)$, or $IVALUEV(I)$ contains the value of given flow parameter of the I th component.

Example

Suppose 1st flow parameter of phase 0 of a stream represents the molar flow rate of a component in the stream.

Find total molar flow rate of all components in the stream.

```
DCL MAX_NC EXT;
```

```
DCL XVALUEV (MAX_NC) FLOAT BIN;
```

```
DCL VTV (MAX_NC) FIXED BIN;
```

```
.
```

```
.
```

```
.
```

```
CALL XGET_FPARMACS(PSTREAM,0,1,XVALUEV,VTV,CODE);
```

```
TF =0;
```

```
DO I=1 TO MAX_NC;
```

```
IF VTV (I) ^ = -1
```

```
    THEN TF = TF + XVALUEV (I);
```

```
END;
```

See also Figure C.5

PUT FPARMACS, XPUT FPARMACS, IPUT FPARMACS

Each of these routines stores the values of a given flow parameter of all components in a given phase of a stream. The only difference among the routines is the attribute of the given values.

Usage

```
DCL PUT_FPARMACS ENTRY (PTR, FIXED BIN, FIXED BIN, DIM (*) FLOAT BIN (63),
                        BIT (1));
```

```
DCL XPUT_FPARMACS ENTRY (PTR, FIXED BIN, FIXED BIN, DIM (*) FLOAT BIN
                        (27), BIT (1));
```

```
DCL IPUT_FPARMACS ENTRY (PTR, FIXED BIN, FIXED BIN, DIM (*) FIXED BIN,
                        BIT (1));
```

```
CALL PUT_FPARMACS (PSTREAM, PHASE_NO, PARM_NO, VALUEV, CODE);
```

```
CALL XPUT_FPARMACS (PSTREAM, PHASE_NO, PARM_NO, XVALUEV, CODE);
```

```
CALL IPUT_FPARMACS (PSTREAM, PHASE_NO, PARM_NO, IVALUEV, CODE);
```

where:

PSTREAM is the pointer to the stream's structure (input).

PHASE_NO is the phase number (input).

PARM_NO is the flow parameter number (input).

VALUEV, XVALUEV, and IVALUEV are vectors containing the values of the given flow parameter of all components in the given phase of the stream (input).

VALUEV is in double precision.

XVALUEV is in single precision.

IVALUEV is in integers.

CODE is the error code (output).

Notes

CODE is "on" for the following cases:

- a. PSTREAM is null.
- b. PHASE_NO is invalid (negative or more than the number of phases of the stream.)
- c. PARM_NO is invalid (less than one, or more than the number of flow parameters of the given phase.)
- d. The size of the vector VALUEV, XVALUEV, IVALUEV, or VTV is not equal to the maximum number of components.

Only flow parameters of those components will be stored that are present in the stream. Value type of all assigned parameters will be set to 3 (calculated).

Example

Suppose that the 1st flow parameter of the phase 0 of a stream represents the molar flow rate.

Suppose further that the total molar flow rate (TF) and mole fractions (X) are known. Store individual molar flow rates.

```
DCL MAX_NC EXT;
```

```
DCL X(MAX NC) FLOAT BIN;
```

```
DCL XPUT_FPARMACS ENTRY(PTR, FIXED BIN, FIXED BIN, DIM(*)FLOAT BIN, BIT(1));
```

```
X(*)=X(*)*TF;
```

```
CALL XPUT_FPARMACS (PSTREAM,0,1,X, CODE);
```

See also Figure C.5

C.4 Service Routines Retrieving Other Variables of Interest

GET METH

It returns the method currently being used to estimate a physical property.

Usage

```
DCL GET_METH ENTRY (FIXED BIN, FIXED BIN, BIT (1));
```

```
CALL GET_METH (PROP_NUMBER, METH, CODE);
```

where:

PROP_NUMBER is the property number (input).

METH is the method number in effect (output).

CODE is the error code (output).

Note

CODE is "on" if PROP_NUMBER is invalid.

Example

See Figure C.7

GET ARG

Given the pointer to the arguments structure, it returns the value of a given argument.

Usage

```
DCL GET_ARG ENTRY (PTR, FIXED BIN, FIXED BIN, BIT (1));
```

```
CALL GET_ARG (PARG, ARG_NO, ARGUMENT, CODE);
```

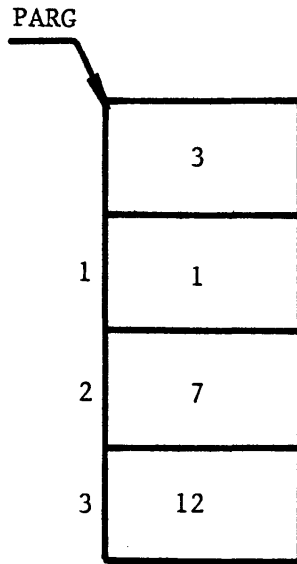
where:

PARG is the pointer to the arguments structure (input).

ARG_NO is the argument number(input).

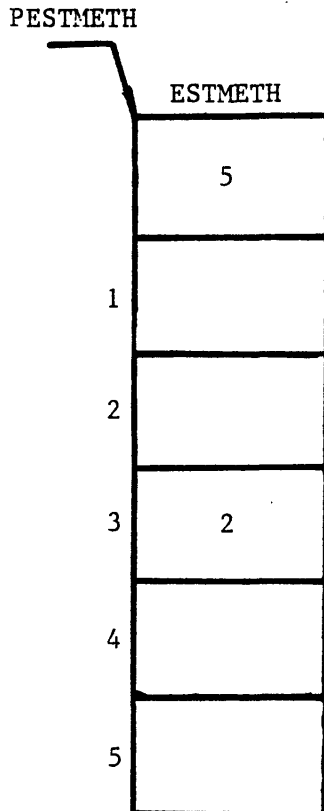
ARGUMENT is the value of the specified argument (output).

CODE is the error code (output).



```
CALL GET_ARG (PARG,2,ARGUMENT,CODE);
/* ARGUMENT=7 */
```

FIGURE C.7 EXAMPLE OF THE USE OF THE SERVICE ROUTINE
RETRIEVING THE ARGUMENTS



```
CALL GET_METH (3,METH);
/* METH=2 */
```

FIGURE C.8 EXAMPLE OF THE USE OF THE SERVICE ROUTINE
RETRIEVING THE PROPERTY ESTIMATION METHODS IN USE

Notes

CODE is "on" for the following cases:

- a. PARG is null
- b. ARG_NO is zero or negative, or greater than the number of arguments.

Example

See Figure C.8

C.5 Service Routines Interacting with the UserGETIN

This routine receives an integer number from the user.

Usage

```
DCL GETIN ENTRY (FIXED BIN, FIXED BIN);
```

```
CALL GETIN (NUMBER, LIMIT)
```

where:

NUMBER is the returned integer number (output).

LIMIT is the upper limit of the number (input).

Notes

The TBS program before calling this routine should write a message on the user's terminal requesting the required INPUT. The GETIN routine then receives and examines the user's response. If the user's response is valid it returns to the caller, otherwise, informs the user and receives a new response and repeats this process until the user's response is acceptable. A valid response is an unsigned integer number up to five digits and less than the specified upper limit.

Example

The TBS Program:

```
PUT SKIP EDIT("ENTER NUMBER OF ITERATIONS:")(A);
CALL GETIN (NIT, 300)
```

user's terminal session:

```
ENTER NUMBER OF ITERATIONS: 4A
```

```
***ERROR*** i 16 "4A" is an invalid integer number.
```

A valid integer number consists of 1 to 5 decimal digits.

```
**reenter the number: 400
```

```
***ERROR*** i 17 the above number must not be greater than 300
```

```
**reenter the number: 40
```

GETLINE

This routine receives a line of characters from the user.

Usage

```
DCL GETLINE ENTRY (CHAR(*),FIXED BIN (21));
CALL GETLINE (LINE,NREAD);
```

where:

LINE is the returned response (output).

NREAD is the number of characters read (output).

Notes

This routine is not usually used for tbs programs.

Example

TBS program:

```
DCL TEXT CHAR(10);
PUT SKIP EDIT ("ENTER TEXT:")(A);
CALL GETLINE (TEXT, NREAD);
```

user's terminal session:

```
ENTER TEXT: Once upon a time
```

ERROR i 1 bad input or line greater than 10 characters.

**reenter the line: AB E

GETRN

This routine receives a real number from the user.

Usage

```
DCL GETRN ENTRY (FLOAT BIN(63));
```

```
CALL GETRN(NUMBER);
```

where:

NUMBER is the returned number (output).

Notes

The TBS Program before calling this routine should write a message on the user's terminal requesting the required input. The GETRN routine then receives and examines the user's response. If the user's response is a valid number it returns to the caller, otherwise it informs the user and receives a new response and repeats this process until the user's response is acceptable.

Example

The TBS Program:

```
PUT SKIP EDIT ("Enter an initial value for T:") (A);
```

```
CALL GETRN (TEMP);
```

User's Terminal Session:

```
Enter an initial value for T:23.4F2
```

```
***ERROR*** i 18 "23.4F2" is an invalid number.
```

```
**Reenter the Number: 23.4e2
```

GET RESPONSE

This routine receives a yes or no answer from the user.

Usage

```
DCL GET_RESPONSE RETURNS (BIT(1));
```

```
FLAG = GET_RESPONSE( );
```

where:

FLAG indicates the user's response (output). It is "on" if the user's response is "yes", and it is "off" if the user's response is "no".

Example

TBS Program:

```
PUT SKIP EDIT ("DO YOU WANT TO USE THE SHORTCUT METHOD?") (A);
```

```
IF GET_RESPONSE( )
```

```
THEN...
```

```
ELSE...
```

User's Terminal Session:

```
DO YOU WANT TO USE THE SHORTCUT METHOD? WHAT
```

```
***ERROR*** i 19 Your response should be yes or no.
```

```
**REENTER PLEASE: yes
```

C.6 Service Routines Performing Arithmetic Operations on Two Parameter SetsPARMS_OPERATOR1

This routine multiplies each parameter in a set of parameters by a given constant and adds to it another given constant. The results are stored in a data structure of comparable size corresponding to another parameter set: $P1_i = P2_i A+B$ for all i

Usage

```
DCL PARMS_OPERATOR1 ENTRY (PTR, PTR, FLOAT BIN(63), FLOAT BIN(63), BIT(1));
```

```
CALL PARMS_OPERATOR1 (PPARM1, PPARM2, A,B, CODE);
```


where:

PPARM1 is the pointer to the target parameters structure (input).

PPARM2 is the pointer to the other parameters structure (input).

A is the multiplication constant (input).

B is the addition constant (input).

CODE is the error code (output).

Notes

CODE is "on" for the following cases:

- a. PPARM1, or PPARM2 is null.
- b. The numbers of parameters of two parameters structure are not the same.
- c. One or more parameters of set pointed by PPARM2 are unspecified.

The value types of all assigned parameters will be set to 3 ("calculated").

Example

See Figure C.9.

PARMS_OPERATOR1V

This routine functions almost the same as PARMS_OPERATOR1 Routine.

The only difference is that the multiplication and addition constants are vectors in this case: $P1_i = P2_i A_i + B_i$ for all i .

Usage

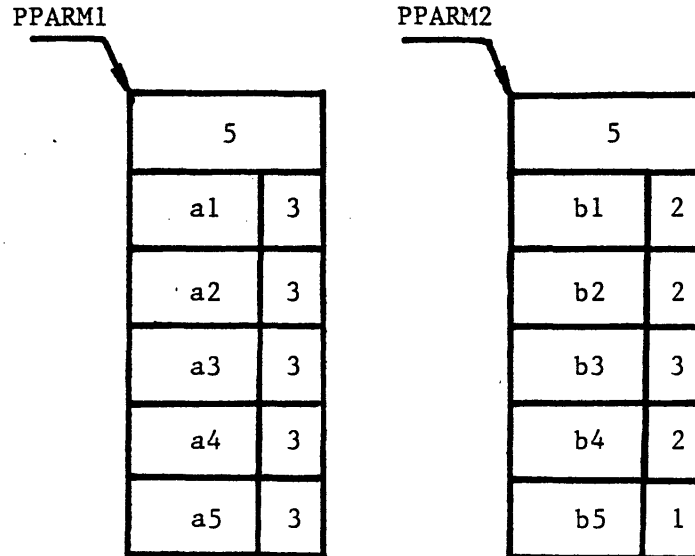
```
DCL PARMS_OPERATOR1V ENTRY (PTR, PTR, DIM(*) FLOAT BIN(63), DIM(*) FLOAT
    BIN(63), BIT(1));
```

```
CALL PARMS_OPERATOR1V (PPARM1, PPARM2, AVEC, BVEC, CODE);
```

where:

AVEC is the vector of multiplication constants (input).

BVEC is the vector of addition constants (input).

PARMS_OPERATOR1

```
CALL PARMS_OPERATOR1 (PPARM1,PPARM2,A,B,CODE);
```

```
/* a1=b1*A+B, a2=b2*A+B, a3=b3*A+B, ... */
```

PARMS_OPERATOR1V

```
CALL PARMS_OPERATOR1V (PPARM1,PPARM2,AVEC,BVEC,CODE);
```

```
/* a1=b1*AVEC(1)+BVEC(1), a2=b2*AVEC(2)+BVEC(2), ... */
```

PARMS_OPERATOR2

```
CALL PARMS_OPERATOR2 (PPARM1,PPARM2,A,B,CODE);
```

```
/* a1=a1+b1*A+B, a2=a2+b2*A+B, ... */
```

PARMS_OPERATOR2V

```
CALL PARMS_OPERATOR2V (PPARM1,PPARM2,AVEC,BVEC,CODE);
```

```
/* a1=a1+b1*AVEC(1)+BVEC(1), a2=a2+b2*AVEC(2)+BVEC(2), ... */
```

FIGURE C.9 EXAMPLES OF THE USE OF SERVICE ROUTINES PERFORMING ARITHMETIC OPERATIONS ON TWO PARAMETER SETS

The other variables are as defined for PARM_OPERATOR1.

Notes

CODE is "on" for the following cases:

- a. PPARAM1, or PPARAM2 is null.
- b. The two parameters structures do not have the same number of parameters.
- c. One or more parameters of set pointed by PPARAM2 are unspecified.
- d. The size of vector AVEC, or BVEC is not the same as the number of parameters.

The value types of assigned parameters will be set to 3 ("calculated").

Example

```
DCL (AVEC(3), BVEC(3)) FLOAT BIN(63);
DCL PARM_OPERATOR1V (PTR, PTR, DIM(*) FLOAT BIN(63), DIM(*) FLOAT
    BIN(63), BIT(1));
BVEC(*) = 0;
AVEC(1) = 1/2;
AVEC(2) = 1/3;
AVEC(3) = 1/4;
CALL PARM_OPERATOR1V (PPARM1, PPARAM, AVEC, BVEC, CODE);
AVEC(1) = 1/2;
AVEC(2) = 2/3;
AVEC(3) = 3/4;
CALL PARM_OPERATOR1V (PPARM2, PPARAM, AVEC, BVEC, CODE);
See also Figure C.9.
```

PARMS_OPERATOR2

This routine multiplies each parameter in a set of parameters by a

given constant and adds to it another given constant. The results are added to another set of parameters and are stored as the new values of this set of parameters: $P1_i = P1_i + P2_i A + B$ for all i .

Usage

```
DCL PARM_OPERATOR2 ENTRY (PTR, PTR, FLOAT BIN(63), FLOAT BIN(63), BIT(1));
CALL PARM_OPERATOR2 (PPARM1, PPARM2, A,B, CODE);
```

where:

The variables are as defined for PARM_OPERATOR1 routine.

Notes

CODE is "on" for the following cases:

- a. PPARM1, or PPARM2 is null.
- b. The number of parameters of two parameter sets are not the same.
- c. One or more parameters of either set are unspecified.

The value types of assigned parameters will be set to 3("calculated").

Example

Add one set of parameters to another set of parameters.

```
CALL PARM_OPERATOR2 (PPARM1, PPARM2, 1,0, CODE);
```

See also Figure C.9.

PARMS_OPERATOR2V

This routine functions almost the same as routine PARM_OPERATOR2.

The only difference is that the multiplication and addition constants are vectors in this case: $P1_i = P1_i + P2_i A_i + B_i$ for all i .

Usage

```
DCL PARM_OPERATOR2V (PTR, PTR, DIM(*) FLOAT BIN(63), DIM(*) FLOAT BIN(63),
    BIT(1));
```

CALL PARMS_OPERATOR2V (PPARM1, PPARAM2, AVEC, BVEC, CODE);

where:

The variables are as defined for routine PARMS_OPERATOR1V.

Notes

CODE is "on" for the following cases:

- a. PPARAM1, or PPARAM2 is null.
- b. The numbers of parameters of two sets are not the same.
- c. The size of the vector AVEC, or BVEC is not the same as the number of parameters.
- d. One or more parameters of either set are unspecified.

The value types of all assigned parameters will be set to 3 ("calculated").

Example

See Figure C.9.

FPARAMCS_OPERATOR1

It assigns a value to a given flow parameter for each component in a given phase of a stream equal to the product of a constant and another flow parameter of the same component in a phase of another stream and added to another constant:

$$F1_i = F2_i * A + B \quad \text{for all } i \text{ (components)}$$

Usage

```
DCL FPARAMCS_OPERATOR1 ENTRY (PTR, FIXED BIN, FIXED BIN, PTR, FIXED BIN,
    FIXED BIN, FLOAT BIN(63), FLOAT BIN(63), BIT(1));
CALL FPARAMCS_OPERATOR1 (PSTREAM1, PHASE_NO1, PARM_NO1, PSTREAM2,
    PHASE_NO2, A, B, CODE);
```

where:

PSTREAM1 is the pointer to the target stream (input).

PSTREAM2 is the pointer to the other stream (input).

PHASE_NO1 is the phase number of the target stream (input).

PHASE_NO2 is the phase number of the other stream (input).

PARM_NO1 is the flow parameter number of the given phase of the target stream (input).

PARM_NO2 is the flow parameter number of the given phase of the other stream (input).

A is the multiplication constant (input).

B is the addition constant (input).

CODE is the error code (output).

Notes

CODE is "on" for the following cases:

- a. PSTREAM1, or STREAM2 is null.
- b. PHASE_NO1 or PHASE_NO2 is invalid.
- c. PARM_NO1, or PARM_NO2 is invalid.
- d. Both streams do not have the same components.
- e. The specified flow parameter of a component in the given phase of the stream pointed by PSTREAM2 is unspecified.

The value types of assigned parameters will be set to 3("calculated").

Example

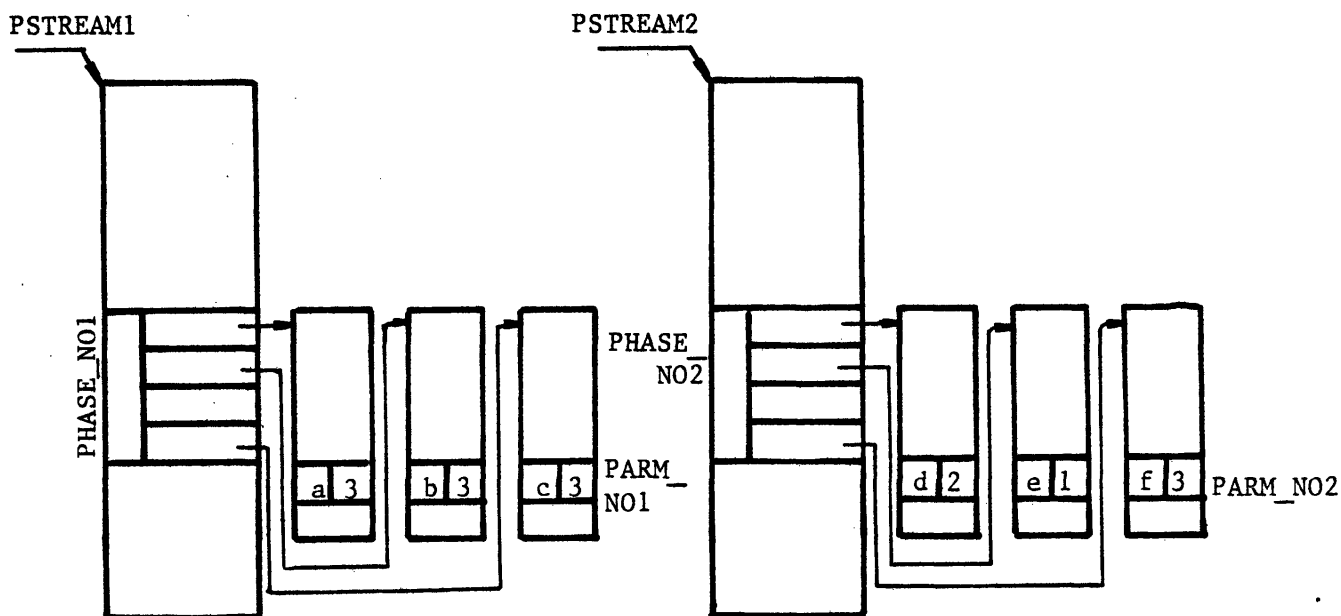
The first flow parameter of phase 0 of each of the streams pointed by PSTREAM1, PSTREAM2 or PSTREAM3 represents the molar flow rate of a component in that stream.

Given the splitting ratio, R, split the stream pointed by PSTREAM1 to streams pointed by PSTREAM2 and PSTREAM3.

```
CALL FPARMACS_OPERATOR1 (PSTREAM2, 0,1, PSTREAM1, 0,1, R,0, CODE);
```

```
CALL FPARMACS_OPERATOR1 (PSTREAM3, 0,1, PSTREAM1, 0,1, 1-R, 0, CODE);
```

See also Figure C.10.



FPMACCS OPERATOR1

```
CALL FPMACCS_OPERATOR1 (PSTREAM1,PHASE_NO1,PARAM_NO1,PSTREAM2,PHASE_NO2,
                        PARAM_NO2,A,B,CODE);
```

```
/* a=d*A+B, b=e*A+B, c=f*A+B */
```

FPMACCS OPERATOR1V

```
CALL FPMACCS_OPERATOR1V (PSTREAM1,PHASE_NO1, PARAM_NO1,PSTREAM2,PHASE_NO2,
                        PARAM_NO2,AVEC,BVEC,CODE);
```

```
/* a=d*AVEC(1)+BVEC(1), b=e*AVEC(2)+BVEC(2), c=f*AVEC(4)+BVEC(4) */
```

FPMACCS OPERATOR2

```
CALL FPMACCS_OPERATOR2 (PSTREAM1,PHASE_NO1,PARAM_NO1,PSTREAM2,PHASE_NO2,
                        PARAM_NO2,A,B,CODE);
```

```
/* a=a+d*A+B, b=b+e*A+B, c=c+f*A+B */
```

FPMACCS OPERATOR2V

```
CALL FPMACCS_OPERATOR2V (PSTREAM1,PHASE_NO1,PARAM_NO1,PSTREAM2,PHASE_NO2,
                        PARAM_NO2,AVEC,BVEC,CODE);
```

```
/* a=a+d*AVEC(1)+BVEC(1), b=b+e*AVEC(2)+BVEC(2), c=c+f*AVEC(4)+BVEC(4) */
```

FIGURE C.10 EXAMPLE OF THE USE OF SERVICE ROUTINES
PERFORMING ARITHMETIC OPERATIONS ON FLOW PARAMETERS OF TWO STREAMS

FPARMACS_OPERATOR1V

This routine functions almost the same as routine FPARMACS_OPERATOR1. The only difference is that the multiplication and addition constants are vectors in this case:

$$F1_i = F2_i * A_i + B_i \text{ for all } i \text{ (components).}$$

Usage

```
DCL FPARMACS_OPERATOR1V ENTRY (PTR, FIXED BIN, FIXED BIN, PTR, FIXED BIN,
    FIXED BIN, DIM(*) FLOAT BIN(63), DIM(*) FLOAT BIN(63), BIT(1));
CALL FPARMACS_OPERATOR1V (PSTREAM1, PHASE_NO1, PARM_NO1, PSTREAM2,
    PHASE_NO2, PARM_NO2, AVEC, BVEC, CODE);
```

where:

AVEC is the vector of the multiplication constants (input).

BVEC is the vector of the addition constants (input).

The other variables are as defined for routine FPARMACS_OPERATOR1.

Notes

CODE is "on" for the following cases:

- a. PSTREAM1, or PSTREAM2 is null.
- b. PHASE_NO1 or PHASE_NO2 is invalid.
- c. PARM_NO1, or PARM_NO2 is invalid.
- d. Both streams do not have the same components.
- e. The size of vector AVEC or BVEC is not equal to the maximum number of components (MAX_NC).
- f. The given flow parameter of a component in the given phase of the stream pointed by PSTREAM2 is unspecified.

The value types of assigned parameters will be set to 3("Calculated").

Example

Phase 1 is the vapor phase and Phase 2 is the liquid phase, and the first flow parameter of each phase is the mole fraction. KVAL contains K values. Liquid phase compositions are already set. Set vapor phase compositions.

```
DCL KVAL(MAX_NC) FLOAT BIN(63);
```

```
DCL BVEC(MAX_NC) FLOAT BIN(63);
```

```
BVEC(*) = 0;
```

```
CALL FPARMACS_OPERATOR1V (PSTREAM1,1,1,PSTREAM1,2,1,KVAL,BVEC,CODE);
```

See also Figure C.10.

FPARMACS_OPERATOR2

It adds to a given flow parameter of each component in a phase of a stream a value equal to the product of a constant and another given flow parameter of the same component in a phase of another stream and added to another constant:

$$F1_i = F1_i + F2_i * A+B \text{ for all } i \text{ (components)}$$

Usage

```
DCL FPARMACS_OPERATOR2 ENTRY (PTR, FIXED BIN, FIXED BIN, PTR, FIXED BIN,  
FIXED BIN, FLOAT BIN(63), FLOAT BIN(63), BIT(1));
```

```
CALL FPARMACS_OPERATOR2 (PSTREAM1, PHASE_NO1, PARM_NO1, PSTREAM2,  
PHASE_NO2, PARM_NO2, A, B, CODE);
```

where:

The variables are as defined for FPARMACS_OPERATOR1 routine.

Notes

CODE is "on" for the following cases:

- a. PSTREAM1, or PSTREAM2 is null.

- b. PHASE_NO1, or PHASE_NO2 is invalid.
- c. PARM_NO1, or PARM_NO2 is invalid.
- d. Both streams do not have the same components.
- e. Flow parameter "PARM_NO2" of a component in phase "PHASE_NO2" of the stream pointed by "PSTREAM2" is unspecified.
- f. Flow parameter "PARM_NO1" of a component in phase "PHASE_NO1" of the stream pointed by "PSTREAM1" is unspecified.

The value types of assigned parameters will be set to 3 ("calculated").

Example

STREAM3 is the result of adding STREAM1 and STREAM2. All streams are of the same type and the first flow parameter of Phase 0 represents the molar flow rate of each component.

```
CALL FPARMACS_OPERATOR1 (PSTREAM3, 0, 1, PSTREAM1, 0, 1, 1.0, 0, CODE);
```

```
CALL FPARMACS_OPERATOR2 (PSTREAM3, 0, 1, PSTREAM2, 0, 1, 1.0, 0, CODE);
```

See also Figure C.10.

FPARMACS_OPERATOR2V

This routine functions the same as FPARMACS_OPERATOR2 with the exception of the addition and multiplication constants that are vectors in this case:

$$F1_i = F1_i + F2_i * A_i + B_i \text{ for all } i \text{ (component).}$$

Usage

```
DCL FPARMACS_OPERATOR2V ENTRY (PTR, FIXED BIN, FIXED BIN, PTR, FIXED BIN,
```

```
    FIXED BIN, DIM(*) FLOAT BIN(63), DIM(*) FLOAT BIN(63), BIT(1));
```

```
CALL FPARMACS_OPERATOR2V (PSTREAM1, PHASE_NO1, PARM_NO1, PSTREAM2,
```

```
    PHASE_NO2, PARM_NO2, AVEC, BVEC, CODE);
```

where:

The symbols are as described for FPARMACS_OPERATOR1V.

Notes

CODE is "on" for the following cases:

- a. PSTREAM1, or PSTREAM2 is null.
- b. PHASE_NO1, or PHASE_NO2 is invalid.
- c. PARM_NO1, or PARM_NO2 is invalid.
- d. Both streams do not have the same components.
- e. The size of vector AVEC or BVEC is not equal to MAX_NC.
- f. Flow parameter "PARM_NO2" of a component in the phase "PHASE_NO2" of the stream pointed by "PSTREAM2" is unspecified.
- g. Any of the target flow parameters are unspecified.

The value types of assigned parameters will be set to 3("calculated").

Example

See Figure C.10.

APPENDIX D

PROCESS ENGINEERING LANGUAGE - DETAILED DESCRIPTION

D.1 COMMAND ELEMENTS

There are very few restrictions in the format of PEL commands. Consequently, instructions can be written without consideration of special coding forms. As long as each command is started on a new line and terminated by a semi-colon (;), the format is completely free. Characters entered after the ";" are ignored.

A command is constructed from symbols.

A symbol is the string of one or more characters.

Character Set

Any character set may be used to write a PEL command. In PEL any character set will be classified according to: Digits, extended alphabetic characters, and special characters.

Digits

There are ten digits (0 through 9).

Special Characters

There are 21 special characters, as shown in Table D.1. These characters perform special functions and all of them are delimiters separating elements of a statement (break characters). The point (.) when used as a part of a number is not treated as a delimiter. Special characters are combined to create other symbols. For example, <= means less than or equal to, ** means exponentiation. Blanks are not allowed in such composite symbols.

Extended Alphabetic Characters

All other characters fall in this category.

TABLE D.1
SPECIAL CHARACTERS

<u>NAME</u>	<u>CHARACTER</u>
Blank	
Equal sign or assignment symbol	=
Plus sign	+
Minus sign	-
Asterisk or multiply symbol	*
Slash or divide symbol	/
Left parenthesis	(
Right parenthesis)
Comma	,
Point or period	.
Single quotation mark or apostrophe	'
Double quotation marks	"
Semicolon	;
Colon	:
"Not" symbol	^
"And" symbol	&
"Or" symbol	
"Greater than" symbol	>
"Less than" symbol	<
Currency symbol or dollar sign	\$
Tab	

Alphanumeric Characters

An alphanumeric character is either an extended alphabetic character or a digit but not a special character.

Symbols

A command is constructed from symbols. There are three types of symbols: Literals, terminal symbols (operators), and identifiers.

Literals

Literals can be grouped into 2 classes:

a) Real numbers. A real number can be in three forms:

1. Without decimal point (e.g. 123)
2. With decimal point (e.g. 123., 123.495)
3. Scientific notation (e.g. 12.3e+1)

A real number can be up to 16 characters including point (.) and e. Exponent can not exceed three digits and must be greater than -129 and less than 128. No embedded blanks are allowed (e.g. 12.3e -4 is invalid). All real numbers may have a "+" or "-" prefix.

b) Integer numbers. An integer number consists of up to 5 digits and no "+" or "-" prefix is allowed.

Terminal Symbols (operators)

The special characters (except the blank and tab) and some combinations of them are terminal symbols. The terminal symbols perform specific functions in a program.

Many of them function as operators.

There are three types of operators: arithmetic, comparison, and Boolean.

The arithmetic operators are:

+ denoting addition or prefix plus

- denoting subtraction or prefix minus
- * denoting multiplication
- / denoting division
- ** denoting exponentiation

The comparison operators are:

- > denoting "greater than"
- >= denoting "greater than" or "equal to"
- = denoting "equal to"
- ^= denoting "not equal to"
- <= denoting "less than or equal to"
- < denoting "less than"

The Boolean operators are:

- & denoting "and"
- | denoting "or"

Table D.2 shows some of the functions of other special characters.

Identifiers

All other symbols are identifiers. An identifier is a single alphabetic character or a string of alphanumeric characters, preceded and followed by a blank or some other delimiter (break-characters).

An identifier can also be enclosed in quotation marks. Such an identifier can contain any character including special characters (except the tab and quotation mark itself which indicates the end of the identifier). The right most blanks in an enclosed identifier are ignored. For example, Identifiers: "AB ", "AB" and AB are exactly the same, but "A B", " AB", and "AB" are 3 different identifiers. The dimension field is a special type of identifier which is enclosed in a pair of single quotation marks and can contain any special characters except the single

TABLE D.2

FUNCTIONS OF OTHER SPECIAL CHARACTERS

NAME	CHARACTER	USAGE
Point or period	.	Indicates decimal point; connects elements of a composite identifier
Currency symbol	\$	Encloses a comment
Colon	:	Currently not used
Comma	,	Separates elements of a list
Semicolon	;	Terminates a command
Assignment	=	Indicates assignment ¹
Blank		Separates elements of a statement
Tab		Separates elements of a statement
Single quotation mark	'	Encloses dimension (units of measurement) of a value
Parentheses	()	Encloses lists; delimits portions of a computational expression
Double quotation marks	"	Encloses an identifier which may contain special characters

¹ Note that the character = can be used as an equal sign and as an assignment symbol.

quotation mark (apostrophe) itself and tab. The rightmost blanks in the dimension field are also ignored. The length of an identifier (not including " or ' of an enclosed identifier) must not exceed 16 characters.

There are no other restrictions on the format of an identifier as long as it cannot be interpreted as a number. For example, 1A, 123B, AB2, 137E1475, "123", "+", and "ABC" all are legal identifiers. The identifiers can be divided into three groups:

1. Language Keywords
2. Established Identifiers
3. User-Supplied Identifiers

Language Keywords

A keyword is an identifier that, when used in proper context, has a specific meaning to the system. A keyword can specify such things as the action to be taken, the nature of data, the purpose of a name. Some keywords can be abbreviated. Note that there is no reserved word in PEL. These identifiers only when used in proper context will be interpreted as keywords. The language keywords can be classified into the following groups:

1. Command Verbs

Each PEL command starts with a keyword called the command verb. The command verb indicates the function of the command.

2. Command Objects

A command object is a keyword representing one of the seven process elements. In some commands the command object follows the command verb to indicate the object upon which the command action should be taken.

3. Phase Fields

A phase field is a keyword representing a specific phase of a stream.

The keyword phasen, or pn may be used to refer to nth phase of a stream. Where n is an integer number. In addition, phase zero which represents the total stream can be also referred by any of these keywords: stream, streams, or s.

4. Parameter References

The parameter may be referenced either by its name or by one of the three following keywords: %parameter_n, %parm_n, or %pn where n is the parameter number.

5. Process Equipment Connection References

A connection may be referenced either by its name or by one of the two following keywords: %cnctn, or %cn where n is the connection number.

6. Print Options

Print options are the keywords used in a print command to specify the type of information to be printed.

7. Profile Parameters

The profile parameters are represented by the following keywords:

sdigit: Significant number of digits

ddigit: Decimal number of digits

dflag: Debugging flag

output: Indicating the output file

input: Indicating the input file

8. Built-In Constants

These are the symbols used to represent built-in constants. All these symbols start with the percent sign "%". These symbols like all other keywords are not reserved words. Such a symbol only will be interpreted as a built-in constants, if there is no user supplied variable with the same name.

9. Built-In Functions

The symbols used to represent built-in functions are another group of language keywords. Such a symbol only will be interpreted as a built-in function, if there is no user created function with the same name.

10. Other Keywords

All other keywords which do not belong to any of the above groups, will fall into this category. Each of these keywords may appear in one or more commands. They may be a part of different clauses of a command. A keyword may belong to more than one of the above groups.

The complete list of keywords and their abbreviations is given in Table D.3.

Established Identifiers

Established identifiers are the keywords of the attached TBS. These identifiers have been set in the TBS by its responsible system administrator. These identifiers can be classified into the following groups:

- 1) Unit types: a unit type is an identifier representing the type of a unit. Symbol "all" is excluded as a unit type.
- 2) Stream types: A stream type is an identifier representing the type of a stream. The symbol "all" is excluded as a stream type.
- 3) Component types: A component type is an identifier representing the type of a component. The symbols "all" and "none" are excluded as component types.
- 4) Pre-defined Function types: A pre-defined function type is an identifier representing the type of a pre-defined function. The symbol "all" is excluded as a function type.

Table D.3

Language Keywords

<u>Group</u>	<u>Keyword</u>	<u>Alternative</u>	<u>Abbreviation</u>
1) Command Verbs	assume		a
	bugs		
	calculate		calc
	clear		clr
	close		cl
	connect		cnct
	continue		ct
	copy		
	create		crt
	delete		del
	deletef		delf
	disconnect		disc
	end		
	escape		esc
	help		h
	include		inc
	leave		lv
	let		
	leta		
	list		l
listf		lf	
listt		lt	
load		ld	
loop		lp	

Table D.3 Continued

<u>Group</u>	<u>Keyword</u>	<u>Alternative</u>	<u>Abbreviation</u>
	news		
	open		
	print		p
	printf		pf
	printt		pt
	profile		prof
	read		rd
	reada		rda
	repeat		r
	save		sv
	specify		sp
	stop		
	terminate		term
	unspecify		unsp
	use		
2) Command Objects	unit	units	u
	stream	streams	s
	component	components	c
	flow		f
	function	functions	fn
	variable	variables	v
	process	processes	pr
3) Phase Fields	phasen		pn
(note: n is the phase number.)	stream	streams	s

Table D.3 Continued

<u>Group</u>	<u>Keyword</u>	<u>Alternative</u>	<u>Abbreviation</u>
4) Parameter References			
(note: n is the parameter number.)	%parametern	%parmn	%pn
5) Connection References			
(note: n is the connection number.)	%cnctn		%cn
6) Print Options	assumed		a
	specified		sp
	unspecified		unsp
	calculated		calc
	parameters		p
	connections		cnct
	type		
	all		
	streams	stream	s
	functions	function	fn
	components	component	c
	flowsheet		
7) Profile Parameters			
	sdigit		
	ddigit		
	dflag		
	output		
	input		

Table D.3 Continued

<u>Group</u>	<u>Keyword</u>	<u>Alternative</u>	<u>Abbreviation</u>
8) Built-In Constants			
	%e		
	%f		
	%n		
	%pi		
	%r		
9) Built-In Functions			
	abs		
	acos		
	asin		
	atan		
	atand		
	atanh		
	cos		
	cosd		
	cosh		
	erf		
	erfc		
	exp		
	log		
	log2		
	log10		
	max		
	min		
	mod		

Table D.3 Continued

<u>Group</u>	<u>Keyword</u>	<u>Alternative</u>	<u>Abbreviation</u>
	sign		
	sin		
	sind		
	sinh		
	sqrt		
	tan		
	tand		
	tanh		
10) Other Keywords	all		
	at		
	brief		bf
	by		
	commands		
	ctlinfo		
	default		
	dimension		
	file		
	from		
	for		
	if		
	notation		
	override		
	print		p
	private		
	property		

Table D.3 Continued

<u>Group</u>	<u>Keyword</u>	<u>Alternative</u>	<u>Abbreviation</u>
	public		
	to		
	type		
	vsctable		
	while		

- 5) Physical Dimensions (Dimensions): A dimension is an identifier enclosed in a pair of single quotation marks representing the units of measurement.
- 6) Physical Property Names: A TBS may allow the user to choose the methods for estimating some or all of the physical properties. Physical property names are identifiers referring to these properties.
- 7) Unit Parameter Names: A unit parameter name is an identifier representing a parameter of a unit type. The symbol "all" and symbols starting with "%" are excluded as parameter names. A user may refer to a parameter either by its name or by language keywords: %parameter_n, %parm_n, or % p_n where n is the parameter number. The symbol "all" refers to all parameters.
- 8) Unit Connection Names: A unit connection name is an identifier representing a connection positions of a unit type. The symbol "all" and symbols starting with "%" are excluded as connection names. A user may refer to a connection either by its name or by language keywords: %cnctn, or %cn, where n is an integer number equal to the connection number. Symbol "all" refers to all connections.
- 9) Stream (phase) Parameter Names: For each phase (including phase 0 which represents the total stream) of each stream type, the identifiers representing the phase parameters are the phase parameter names of that phase of that stream type.
- 10) Flow Parameter Names: For each phase (including phase 0 which represents the total stream) of each stream type the identifiers representing the flow parameters are the flow parameter names of that phase of that stream type.
- 11) Component Parameter Names: For each component type the identifiers

representing the parameters are the component parameter names of that component type.

12) Pre-defined Function Parameter Names: For each pre-defined function type the identifiers representing the parameters (coefficients) are the pre-defined function parameter names of that function type.

User-Supplied Identifiers

These are the identifiers provided by the user to identify the objects or files he creates. The user-supplied identifiers are as follows:

- 1) Unit identifiers: when a user creates a unit, he must assign an identifying name to that unit.
- 2) Stream-identifiers: when a user creates a stream, he must assign an identifying name to the stream.
- 3) Component identifiers: are the names of the pure components.
- 4) Function-identifiers: when a user creates a function, he must assign an identifying name to the function.
- 5) Process-identifiers: when a user attempts to save or to load a process he must identify it by a name.
- 6) Simple Variable identifiers: the user must provide a name for a simple variable to represent its value. Although there is no restriction on the format of these identifiers it is recommended that such identifiers do not start with percent sign (%) so that the user can distinguish them from the built-in constants.
- 7) Component File Names: these are the names of component public or private files. A user may create any number of component private files. A TBS may have any number of component public files. Users may share their component files.
- 8) Process File Names: these are the names of process files. A user may

have any number of process files, each containing any number of processes. Users may share their process files.

Composite Identifiers

These are the identifiers composed of two or more identifiers, and separated by points (.). Each composing identifier is called an element. Each element may be a language keyword, an established identifier, or a user-supplied identifier. No embedded blanks are allowed between the elements. Currently, the qualified variables are the only composite identifiers. Qualified variables are those variables referring to a parameter of a unit, stream, component, or a pre-defined function. They are as follows:

Unit Qualified Variables

A unit qualified variable refers to a parameter of a unit. The general format of a unit qualified variable is:

unit.unit-identifier.parameter

For example: variable u.a.t refers to parameter "t" of unit "a".

Component Qualified Variables

A component qualified variable refers to a parameter of a component. The general format of a component qualified variable is:

component.component-identifier.parameter

For example: c.co2.%p3 refers to third parameter of component co2.

Stream Qualified Variables

A stream qualified variable refers to a parameter of a specific phase of a stream. The general format of a stream qualified variable is:

stream.stream-identifier.phase-field.parameter

For example: variable s.feed.p1.t refers to parameter "t" of phase one of stream "feed".

Flow Qualified Variables

A flow qualified variable refers to a flow parameter of a component in a specific phase of a stream. The general format of a flow qualified variable is:

flow.stream-identifier.phase-field.component-identifier.flow-parameter

For example: variable f.feed.p1.co2.x refers to flow parameter "x" of component "co2" in phase one of stream "feed".

Function Qualified Variables

A function qualified variable refers to a parameter (coefficient) of a pre-defined function. The general format of a function qualified variable is:

function.function-identifier.parameter

For example: variable fn.h.ao refers to parameter "ao" of pre-defined function "h".

Blanks

Blanks may be used freely in a command. One or more blanks must be used to separate identifiers and constants that are not separated by some other delimiters or by a comment.

Comments

Comments are permitted whenever blanks are allowed in a command. A comment is treated as a blank and can therefore be used in place of a required separating blank. Comments do not otherwise affect execution of a command; they are used only for documentation purposes. The general format of a comment is:

\$ character string \$

The comment itself may contain any character except \$, which would be interpreted as terminating the comment. A comment may not continue to the next line.

D.2 EXPRESSIONS

An expression is a representation of a value. A single constant, a simple variable, or a qualified variable is an expression. Combinations of constants and/or variables, along with operators and/or parentheses, are expressions. An expression that contains operators is an operational expression. The constants and variables of an operational expression are called operands.

Examples of expressions are:

27

work

(x-y)*u.heat.A

Use of Expressions

Expressions may appear in a PEL statement for the following purposes:

- 1) To represent a value

examples are:

repeat for y from (A+B) to (A*B);

specify unit (heat) (A=x*B, U=u.heat.A*C);

- 2) To define a function

for example:

create function (h(x,y)=x*y*+sqrt(x+y));

- 3) To be used in while and if clauses for checking conditions.

examples are:

repeat for Y from (1) to (10) while (A<B&C>D);

specify v(x=10) if (Y>Z);

Expression Operations

An operational expression can specify one or more single operations. The class of operation is dependent upon the class of operator specified

for the operation. There are three classes of operations: arithmetic, boolean, and comparison.

Arithmetic Operations

An arithmetic operation is one that is specified by combining operands with one of the following operators:

+ - * / **

The plus sign and minus sign can appear either as prefix operators (associated with and preceding a single operand, such as +A or -A) or as infix Operators (associated with and between two operands, such as A+B or A-B). All other arithmetic operators can appear only as infix operators.

An expression of greater complexity can be composed of a set of such arithmetic operations. Note that prefix operators can precede and be associated with any of the operands of an infix operation. For example, expression A*-B is equivalent to A*(-B), and A--B is equivalent to A-(-B).

Only one prefix operator can precede and be associated with a single variable. For example, expression A*--B is in error.

Boolean Operations

In PEL a boolean operation is one that is specified by combining operands with one of the following operators:

"and" symbol &

"or" symbol |

Both operators can be used as infix operators only.

Note that there is no boolean data in PEL, arithmetic data is the only data in PEL. The result of a boolean operation is either +1 or -1 depending on the operands.

The result of an "and" operation is +1 only if both operands are positive or zero otherwise the result is -1. The result of an "or"

operation is +1 unless both operands are negative, in which case it is -1.

Comparison Operations

A comparison operation is one that is specified by combining operands with one of the following operators:

"less than"	<
"less than or equal to"	<=
"equal to"	=
"not equal to "	^=
"greater than or equal to"	>=
"greater than"	>

All of these operators can be used as infix operators only. The result of a comparison operation is +1 if the relationship is true, -1 if the relationship is false.

Combinations of Operations

The most common occurrence of comparison operations and boolean operations are in "while" and "if" clauses.

```
For example: repeat for X from (1) to (10)
              while (Y<B&C>D);
```

Comparison operations and boolean operations need not be limited to these statements, however, the following statement could be valid:

```
specify variables (X=A<B,Y=C=D);
```

In this example the value of +1 would be assigned to X if A is less than B, otherwise, the value -1 would be assigned. In the same way the value +1 would be assigned to Y if C is equal to D, otherwise, the value -1 would be assigned. Note that in the Y assignment the first "=" sign is the assignment symbol; the second "=" sign is the comparison operator.

Different types of operations can be combined within the same operational expression. Any combination can be used. For example, the expression shown in the following command is valid:

```
specify variables (A=6,B=-8,C=9,D=-1);
```

```
specify variable (X=A+B<C&D);
```

Each operation within the expression is evaluated according to the rules for that kind of operation.

B would be added to A resulting in the value, -2.

-2 would be compared to C resulting in +1 since the relationship is true.

An "and" operation would be performed as a result of the comparison between (+1) & D resulting in -1.

The result of "and" operation (-1) would be assigned to X.

The expression in this example is described as being evaluated operation-by-operation, from left to right. Such would be the case for this particular expression. The order of evaluation, however, depends upon the priority of the operators appearing in the expression.

Similarly, "while" and "if" expressions can have combined types of operations. The "while" and "if" expressions would be evaluated like any other expression. If the result of expression is positive or zero, the relationship would be considered to be true (conditions hold). If the result of the expression is negative, the relationship would be considered false (conditions do not hold).

For example:

```
calculate units (heater, turbine) if (u.Heat.A<100);
```

If parameter A of unit Heat is less than 100, the result of expression would be +1 and consequently the command would be executed. Similarly the

following command will be executed if Y+Z is zero or positive.

specify variable (X=10) if (Y+Z);

Priority of Operators

In the evaluation of expressions, priority of the operators is as follows:

prefix +	prefix -	(highest)
**		↓
*		
infix +	infix -	
< <= = ^= >= >		
&		
		(lowest)

If two or more operators of the same priority appear in the same expression, the order or priority of those operators is from left to right; that is, the leftmost operator in the expression is evaluated first. Note that the order of evaluation of the expression in the command:

specify variable (X=A+B<C&D);

is the result of the priority of the operators. It is as if various elements of the expression were enclosed in parentheses as follows:

(A)+(B)
 (A+B)<(C)
 ((A+B)<C)&(D)

The order of evaluation (and, consequently, the result) of an expression can be changed through the use of parentheses. The above expression, for example, might be changed as follows:

(A+B)<(C&D)

In such an expression, those expressions enclosed in parentheses are evaluated first to be reduced to a single value, before they are considered in relation to surrounding operators.

Operands of an Expression

An operand of an expression can be a constant (number), a simple variable, or a qualified variable. An operand can also be an expression that represents a value that is the result of a computation, as shown in the following expression:

$$B*\text{sqrt}(X)$$

In this example, the expression `sqrt(X)` represents a value that is equal to the square root of the value `X`. Such an expression is called a function reference.

The function `sqrt` is one of the PEL built-in functions. There are other types of functions in PEL. A complete discussion of functions follows.

D.3 FUNCTIONS

There are three types of functions available to the user. They are:

pre-defined functions

user-defined functions

built-in functions

The general form of a function reference is as follows:

function-identifier (expression, expression, ...)

Pre-Defined Functions

These are the functions defined and established in the TBS by the responsible system administrator, by providing the appropriate templates, and required subprograms. The user incorporates these functions into his problem solving capability by a "create" command. The coefficients of such a function (if any) may be explicitly provided by a "specify" command or may be assigned by a "calculate" command which in effect calls upon a calculating routine. For example, suppose the TBS has a pre-defined function of type I1 which is in the form:

$$F(X1,X2) = \int_{X1}^{X2} 3X^2 dX$$

The following commands show how the user can use such a function type:

```
**COMMAND: create function (H) type=I1;
```

```
**COMMAND: specify variable (X=H(0,2));
```

```
**COMMAND: print variable (X);
```

```
X=8
```

Suppose there is another pre-defined function of type L1 which is in the form: $F(X)=A0+A1*X$, where $A0$ and $A1$ are function parameters. Suppose the calculating routine for this function when invoked will perform a least square curve fitting on user supplied data to determine parameters

(coefficients) A0 and A1 of the function. The following commands show how the user can use such a function:

```
**COMMAND: create function (P) type=L1; $A0 and A1 are unspecified now $
```

```
**COMMAND: specify function (P)(A0=2,A1=4);
```

```
**COMMAND: specify variable (X=P(5)); $X=2+4*5=22$
```

```
**COMMAND: print v(X);
```

```
      X=22
```

```
**COMMAND: calculate function (P);
```

```
*entering routine leasts for level 1 calculation of function "P"
```

```
**enter number of data:  4
```

```
**enter X and Y:
```

```
  1:  2  10
```

```
  2:  4  20
```

```
  3:  6  30
```

```
  4:  7  35
```

```
**COMMAND: specify variable (X=P(3));
```

```
**COMMAND: print variable (X);
```

```
      X=15
```

The "delete" command can remove the function from the users working area, if it is not referred to in any current user-defined function.

User-Defined Functions

These are the functions that the user may provide at will to improve his problem solving capability. For example:

```
create function (XXX(X,Y,Z)=X**Y+sqrt(Z));
```

```
specify variable (A=XXX(2,1,4)); $A=2** 1+sqrt(4)=4$
```

The "delete" command will remove the function from the users working area, if the function is not referred to in any other current user-defined

function:

```
delete function (XXX);
```

Built-In Functions

These are the functions built-in to the system and available to the user. Each such function will only be interpreted as a built-in function, if there is no other user created function with that name.

The built-in functions can be classified according to the features they are intended to serve. These classes are:

Arithmetic

Mathematical

Arithmetic Built-in Functions

These functions allow the programmer to investigate simple properties of arithmetic values. They are:

abs

max

min

mod

sign

Mathematical Built-In Functions

These functions provide standard mathematical operations. They are:

acos	log
asin	log2
atan	log10
atand	sin
atanh	sind
cos	sinh
cosd	sqrt

cosh	tan
erf	tand
erfc	tanh
exp	

A description of each built-in function is given in this section. They are presented in alphabetical order.

abs(x)

abs returns the absolute value of a given expression x.

acos(x)

acos returns a value that represents the inverse (arc) cosine in radians of a given expression x.

The absolute value of x must be less than or equal to 1, i.e., $\text{abs}(x) \leq 1$. The result is in the range:

$$0 \leq \text{acos}(x) \leq \pi$$

asin(x)

asin returns a value that represents the inverse (arc) sine in radians of a given expression x.

The absolute value of x must be less than or equal to 1, i.e., $\text{abs}(x) \leq 1$. The result is in the range:

$$-\pi/2 \leq \text{asin}(x) \leq \pi/2$$

atan(x1) or atan(x1,x2)

atan returns a value that represents the inverse (arc) tangent in radians of a given value x1 or of a given ratio x1/x2. x1 and x2 are expressions.

If x1 alone is specified the result is in the range:

$$-\pi/2 < \text{atan}(x1) < \pi/2$$

If x1 and x2 are specified, it is an error if x1 and x2 are both zero.

The results for all other values of x_1 and x_2 are given by:

$\arctan(x_1/x_2)$ for $x_2 > 0$

$\pi/2$ for $x_2 = 0$ and $x_1 > 0$

$-\pi/2$ for $x_2 = 0$ and $x_1 < 0$

$\pi + \arctan(x_1/x_2)$ for $x_2 < 0$ and $x_1 \geq 0$

$-\pi + \arctan(x_1/x_2)$ for $x_2 < 0$ and $x_1 < 0$

atand (x1) or atand (x1,x2)

atand returns a value that represents the inverse (arc) tangent in degrees of a given value x_1 or of a given ratio x_1/x_2 . x_1 and x_2 are expressions.

If x_1 alone is specified, the result is in the range:

$-90 < \text{atand}(x_1) < 90$

If x_1 and x_2 are specified, the result is defined in terms of the function atan as:

$180/\pi * \text{atan}(x_1, x_2)$

atanh(x)

atanh returns a value that represents the inverse (arc) hyperbolic tangent of a given expression x .

The absolute value of x must be less than 1, i.e., $\text{abs}(x) < 1$.

cos(x)

cos returns a value that represents the cosine of a given value x . x is an expression whose value is in radians.

cosd(x)

cosd returns a value that represents the cosine of a given value x . x is an expression whose value is in degrees.

cosh(x)

cosh returns a value that represents the hyperbolic cosine of a given expression x .

erf(x)

erf returns a value that represents the error function of a given expression x.

The result is given by:

$$\text{erf}(x) = \frac{2}{\sqrt{\pi}} \int_0^x e^{-t^2} dt$$

erfc(x)

erfc returns a value that represents the complement of the error function of a given expression x.

The result is defined in terms of the function erf as:

$$1 - \text{erf}(x)$$

exp(x)

exp returns a value that represents the base of the natural logarithm system, e, to the power of expression x.

log(x)

log returns a value that represents the natural logarithm, i.e., base e, of a given value x.

x is an expression whose value must be positive.

log2(x)

log2 returns a value that represents the binary logarithm, i.e., base 2, of a given value x.

x is an expression whose value must be positive.

log10(x)

log10 returns a value that represents the common logarithm, i.e., base 10, of a given value x.

x is an expression whose value must be positive.

max(x1,x2)

max returns, from a set of two arguments, the value of the argument with the largest value.

x1,x2 are two expressions from which the largest is to be returned.

min(x1,x2)

min returns, from a set of two arguments, the value of the argument with the smallest value.

x1,x2 are two expressions from which the smallest is to be returned.

mod(x1,x2)

mod returns the smallest positive value, R, such that:

$(x1 - R)/x2 = n$, where n is an integer.

R is the smallest positive value that must be subtracted from a given value x1 to make it exactly divisible by the given value x2.

If x1 is positive, R is the remainder of the division of x1 and x2; if x1 is negative, R is the modular equivalent of this remainder.

x2 must not be zero.

x1, and x2 are expressions.

sign(x)

sign returns a value that indicates whether a given value x is positive, zero, or negative. The value returned is as follows:

value of x	value returned
$x > 0$	+1
$x = 0$	0
$x < 0$	-1

x is an expression.

sin(x)

sin returns a value that represents the sine of a given value x.

x is an expression whose value is in radians.

sind(x)

sind returns a value that represents the sine of a given value x .
 x is an expression whose value is in degrees.

sinh(x)

sinh returns a value that represents the hyperbolic sine of a given expression x .

sqrt(x)

sqrt returns a value that represents the square root of x . The result is the positive square root of x .
 x is a non-negative expression.

tan(x)

tan returns a value that represents the tangent of a given value x .
 x is an expression whose value is in radians.

tand(x)

tand returns a value that represents the tangent of a given value x .
 x is an expression whose value is in degrees.

tanh(x)

tanh returns a value that represents the hyperbolic tangent of a given value x .
 x is an expression whose value is in radians.

D.4 SEMI FORMAL DEFINITION OF PEL SYNTAX

The following conventions and definitions facilitate the description of the syntax of the PEL commands.

BRACKET

[] indicates that the content of the bracket is optional. In other words the contents of the bracket can be repeated none or one time.

[]* Indicates that the content of the bracket can be repeated none or more times.

BRACES

{ } Indicates that only one of the items contained in braces and separated by | can be specified.

UNDERLINED WORDS

The underlined words are TBS pre-established words.

NOT UNDERLINED AND NOT ENCLOSED IN < >.

These are language keywords. They can appear as specified in the syntax notation or can be replaced by their abbreviations or alternatives.

ENCLOSED IN < >

Symbols enclosed in corner-brackets represent syntactic classes whose description follows. In other words, such symbols have to be replaced by other symbols as described next.

<identifier>

It is a user supplied identifier.

<number>

It is a real number.

<integer-number>

It is an integer number up to five digits.

<phase-field>

It represents a specific phase of a stream. Keywords phase0, p0, stream, streams, or s represents phase zero. Keyword phasen, or pn represents the nth phase. n is an integer number. It should be noticed that the number of phases of each stream depends upon the stream type.

<parameter>

It refers to a parameter of a unit, stream, component, or pre-defined function. A parameter can be referred either by its pre-established name or by one of the keywords %parametern, %parmn, or %pn, where n is the parameter number:

{ parameter | %parametern | %parmn | %pn }

<connection>

It refers to a connection position of a unit. A connection can be referred either by its pre-established name, or by one of the keywords %cnctn, or %cn, where n is the connection number:

{ connection | %cnctn | %cn }

<expression>

It is a mathematical expression as described in sections D.2.

<unit>

It is a unit identifier.

<stream>

It is a stream identifier.

<component>

It is a component identifier.

<function>

It is a pre-defined or user-defined function identifier.

<variable>

It is a simple or qualified variable identifier.

<process>

It is a process identifier.

<file>

It is a file identifier.

<unit-list>

It is a list of unit identifiers:

<unit> [, <unit>]*

No duplicate name is allowed.

<stream-list>

It is a list of stream identifiers:

<stream>[,<stream>]*

No duplicate name is allowed.

<component-list>

It is a list of component identifiers:

<component>[,<component>]*

No duplicate name is allowed.

<function-list>

It is a list of function identifiers:

<function>[,<function>]*

Each function can be pre-defined or user defined but can not be built-in.

No duplicate name is allowed.

<variable-list>

It is a list of variable identifiers:

`<variable> [, <variable>]*`

Variables can be simple or qualified.

`<process-list>`

It is a list of process identifiers:

`<process> [, <process>]*`

No duplicate name is allowed.

`<file-list>`

It is a list of file identifiers:

`<file> [, <file>]*`

No duplicate name is allowed.

`<dummy-argument-list>`

It is a list of dummy arguments used in creating a user defined function:

`<identifier> [, <identifier>]*`

No duplicate name is allowed.

`<parameter-list>`

It is a list of parameters:

`<parameter> [, <parameter>]*`

No duplicate name is allowed.

`<connection-list>`

It is a list of unit connection names:

`<connection> [, <connection>]*`

No duplicate name is allowed.

`<flow-parameter-list>`

It is a list of flow parameters of the components present in a stream:

`<component>{all|(<parameter-list>)}`

`[, <component>{all|(<parameter-list>)}]*`

<arguments>

It is a list of integer numbers used in calculate commands to be passed as arguments to the calculating routine:

```
<integer-number>[,<integer-number>]*
```

<transfer-point>

It is a unit, stream, component, or function identifier used in calculate commands:

```
{<unit>|<stream>|<component>|<function>}
```

<parameter-specification-list>

It provides data for the parameters of units, streams, components, or pre-defined functions:

```
Option 1: <parameter>=<expression>['<dimension>']
          [,<parameter>=<expression>['<dimension>']]*
```

```
Option 2: all=[[,]*<expression>['<dimension>']]
          [[,]*<expression> ['<dimension>']]*
```

In the second option it is not necessary to specify all the parameters, but the sequence of the parameters must be preserved by providing the repeated commas. Suppose a unit has 5 parameters 1st and 4th parameters are P and T. The following are valid examples of parameter-specification-lists for the above unit:

```
T=2,%P2=3,P=4 $1st parameter =4, 2nd parameter = 3,
```

```
4th parameter =2 $
```

```
all=,,8,3 $3rd parameter = 8, 4th parameter = 3 $
```

```
all=12,,9 $1st parameter =12, 3rd parameter = 9 $
```

<flow-parameter-specification-list>

It provides data for the flow parameters of the components present in a stream:

`<component>(<parameter-specification-list>)`

`[,<component>(<parameter-specification-list>)]*`

`<object>`

It is a keyword representing a class of objects (command objects):

`{unit|stream|component|flow|function|variable|process}`

`<verb>`

It is a keyword indicating the action to be taken by a command (command verb):

`{assume|bugs|calculate|clear|close|connect|continue|copy|create|delete|
deletef|disconnect|end|escape|help|include|leave|let|leta|list|listf|
listt|load|loop|news|open|print|printf|printt|profile|read|reada|repeat|
save|specify|stop|terminate|unspecify|use}`

`<if-clause>`

Some commands may have if-clause. The execution of such a command depends on the conditions set in the if-clause. The if-clause is in the following form:

`if(<expression>)`

The command having the if-clause will not be executed, if the result of the expression is negative; it will be executed otherwise. Examples of valid if-clauses are:

`if(A>B)`

`if(A>B & C>D)`

`if(A&B)`

`if(-3)`

Table D.4 lists the general format (syntax) of each command.

Table D.4 GENERAL FORMAT OF PEL COMMANDS

Command Verb	Command Object	Command Body	Terminator
assume	unit	(<u><unit-list></u>)(<u><parameter-specification-list></u>) [<u><if-clause></u>]	;
	component	(<u><component-list></u>)(<u><parameter-specification-list></u>) [<u><if-clause></u>]	;
	function	(<u><function-list></u>)(<u><parameter-specification-list></u>) [<u><if-clause></u>]	;
	stream	(<u><stream-list></u>) [<u><phase-field></u>](<u><parameter-specification-list></u>)[[<u><phase-field></u>](<u><parameter-specification-list></u>)]* [<u><if-clause></u>]	;
	flow	(<u><stream-list></u>)[<u><phase-field></u>](<u><flow-parameter-specification-list></u>)[[<u><phase-field></u>](<u><flow-parameter-specification-list></u>)]* [<u><if-clause></u>]	;
	variable	(<u><variable></u> =(<u><expression></u> [<u>'dimension'</u>][, <u><variable></u> =(<u><expression></u> [<u>'dimension'</u>]]*)[<u><if-clause></u>]	;
	bugs		;

Table D.4 continued

Command Verb	Command Object	Command Body	Terminator
calculate	unit	all[[(<arguments>)]] [<if-clause>]	;
		(<unit>[[(<arguments>)[, <transfer-point>]]] [, <unit>[[(<arguments>)[, <transfer-point>]]]]*) [<if-clause>]	;
component		all[[(<arguments>)]] [<if-clause>]	;
		(<component>[[(<arguments>)[, <transfer-point>]]] [, <component>[[(<arguments>)[, <transfer-point>]]]]*) [<if-clause>]	;
function		all[[(<arguments>)]] [<if-clause>]	;
		(<function>[[(<arguments>)[, <transfer-point>]]] [, <function>[[(<arguments>)[, <transfer-point>]]]]*) [<if-clause>]	;
stream		all [[(<arguments>)]][<if-clause>]	;
		(<stream>[[(<arguments>)[, <transfer-point>]]] [, <stream>[[(<arguments>)[, <transfer-point>]]]]*) [<if-clause>]	;
clear			;
close	component	[{public private}] file	;
	process	file	;

Table D.4 Continued

Command Verb	Command Object	Command Body	Terminator
connect	unit	<unit> at <u>connection-name</u> =<stream> [<u>connection-name</u> =<stream>]*	;
		<unit> at all =[[,]*<stream>][[,]*<stream>]*	;
Continue			;
copy	components	{all (<component-list>)}[<u>type=type</u>][<u>override</u>]	;
		file	;
create	unit	(<unit-list>)[<u>type=type</u>]	;
	component	(<component-list>)[<u>type=type</u>]	;
	function	(<function-list>)[<u>type=type</u>]	;
		(<function>(<dummy-argument-list>)=<expression> [,<function>(<dummy-argument-list>)=<expression>]*);	
	stream	(<stream-list>)[<u>type=type</u>]	;
	flow	{all (<stream-list>)}{all (<component-list>)}	;
delete	unit	{all (<unit-list>)}	;
	component	{all (<component-list>)}	;
	function	{all (<function-list>)}	;
	stream	{all (<stream-list>)}	;
	flow	{all (<stream-list>)}{all (<component-list>)}	;

Table D.4 Continued

Command Verb	Command Object	Command Body	Terminator
deletef	component	{all (<component-list>)} [type= <u>type</u>]	;
	process	{all (<process-list>)}	;
disconnect	unit	<unit> at {all (<connection-list>)}	;
end			;
escape			;
help		{commands notation <object> <verb><object> <verb> all}	;
include	component	(<component-list>)[type= <u>type</u>][override]	;
leave		[brife]	;
{let leta}	unit	<unit>=<unit>	;
	component	<component>=<component>	;
	function	<function>=<function>	;
	stream	<stream>=<stream>	;
	flow	<stream>=<stream>	;

Table D.4 Continued

Command Verb	Command Object	Command Body	Terminator
list	unit		;
	component		;
	function		;
	stream		;
	variable		;
listf	component {all [<u>type=type</u>]} [{private public}]		;
	process		;
listt	unit		;
	component		;
	function		;
	stream		;
load	component (<component-list>)[<u>type=type</u>][{public private}]		;
	process <process>		;
loop			;
news			;

Table D.4 Continued

Command Verb	Command Object	Command Body	Terminator
open	component	[{public private}] file (<file>)	;
	process	file(<file>)	;
print	unit	{all (<unit-list>)}{unspecified assumed specified calculated parameters connections type all} [if-clause>]	;
	component	{all (<component-list>)}{unspecified assumed specified calculated parameters streams type all} [<if-clause>]	;
	function	{all (<function-list>)}{unspecified assumed specified calculated parameters functions type all} [<if-clause>]	;
	stream	{all (<stream-list>)}{unspecified assumed specified calculated parameters connections type all} [<if-clause>]	;
flow	{all (<stream-list>)}{unspecified assumed specified calculated parameters components all} [<if-clause>]	;	
variable	{all (<variable-list>)}[<if-clause>]	;	
process	{all flowsheet} [<if-clause>]	;	

Table D.4 Continued

Command Verb	Command Object	Command Body	Terminator
printf	component	{all (<component-list>)}[<u>type=type</u>] [{public private}]	;
printt		{unit stream component function}{[<u>type=type</u>] all} {dimension property}{<integer-number> all} ctlinfo	;
		vsctable	;
		all	;
profile	print	[{sdigit ddigit dflag output input}= <integer-number>]*	;
		default	;

Table D.4 Continued

Command Verb	Command Object	Command Body	Terminator
{read reada}	unit	(<u><unit-list></u>){all (<u><parameter-list></u>)}[<u><if-clause></u>]	;
		all	[<u><if-clause></u>]
component		(<u><component-list></u>){all (<u><parameter-list></u>)} [<u><if-clause></u>]	;
		all	[<u><if-clause></u>]
function		(<u><function-list></u>){all (<u><parameter-list></u>)} [<u><if-clause></u>]	;
		all	[<u><if-clause></u>]
stream		(<u><stream-list></u>)[<u><phase-field></u>]{all (<u><parameter- list></u>)}[[<u><phase-field></u>]{all (<u><parameter-list></u>)}]* [<u><if-clause></u>]	;
		all	[<u><if-clause></u>]
flow		(<u><stream-list></u>)[<u><phase-field></u>]{all (<u><flow-parameter- list></u>)}[[<u><phase-field></u>]{all (<u><flow-parameter- list></u>)}]*[<u><if-clause></u>]	;
		all	[<u><if-clause></u>]
variable		{all (<u><variable-list></u>)} [<u><if-clause></u>]	;

Table D.4 Continued

Command Verb	Command Object	Command Body	Terminator
repeat		for <variable> from (<expression>[' <u>dimension</u> ']) to (<expression>[' <u>dimension</u> '])[by (<expression> [' <u>dimension</u> '])][while (<expression>)]	;
		for <variable>=<expression>[' <u>dimension</u> '] [,<expression>[' <u>dimension</u> ']]*) [while (<expression>)]	;
save	component	{all (<component-list>)} [override]	;
	process	<process>	;
specify	unit	(<unit-list>)(<parameter-specification-list>) [<if-clause>]	;
	component	(<component-list>)(<parameter-specified-list>) [<if-clause>]	;
	function	(<function-list>)(<parameter-specification-list>) [<if-clause>]	;
	stream	(<stream-list>)[<phase-field>](<parameter- specification-list>)[[<phase-field>](<parameter- specification-list>)]* [<if-clause>]	;
	flow	(<stream-list>)[<phase-field>](<flow-parameter- specification-list>)[[<phase-field>](flow-parameter- specification-list>)]* [<if-clause>]	;
	variable	(<variable>=<expression>[' <u>dimension</u> '][,<variable>= <expression>[' <u>dimension</u> ']]*) [<if-clause>]	;

Table D.4 Continued

Command Verb	Command Object	Command Body	Terminator
stop			;
terminate			
	component	file (<file-list>)	;
	process	file (<file-list>)	;
unspecify			
	unit	(<unit-list>) {all (<parameter-list>)}	;
		all	;
	components	(<component-list>){all (<parameter-list>)}	;
		all	;
	functions	(<function-list>){all (<parameter-list>)}	;
		all	;
	streams	(<stream-list>)[<phase-field>]{all (<parameter-list>)} [[<phase-field>]{all (<parameter-list>)}]*	;
		all	;
	flow	(<stream-list>)[<phase-field>]{all (<flow-parameter-list>)} [[<phase-field>]{all (<flow-parameter-list>)}]*	;
		all	;

Table D.4 Continued

Command Verb	Command Object	Command Body	Terminator
		variables {all (<variable-list>)}	;
use	print		;
		[<u>property-name</u> = <integer-number>]*	;
	default		;

D.5 PEL COMMANDS

PEL commands are described in this section. The commands are presented in alphabetical order, and each is accompanied by the following information:

- 1) FUNCTION - a short description of the meaning and use of the command.
- 2) GENERAL FORMAT - the syntax of the command.
- 3) GENERAL RULES - the rules that must be known by the user for proper usage of the command.
- 4) EXAMPLES - one or more examples.

ASSUME COMMANDS

The assume commands are used to supply the estimated numerical values of parameters, when the user is not certain about the parameter values and the system requires such data for initiating calculations (e.g. recycle streams).

The general format and rules of these commands are the same as the specify commands (replace command verb, specify, with assume). Note for simple variables the assume command and specify command are exactly the same.

BUGS COMMAND

FUNCTION: to print reported errors in the system and the attached TBS.

GENERAL FORMAT: bugs;

CALCULATE COMMANDS

These commands are used to perform the calculations associated with an object. The calculate commands can be grouped into the following four commands:

I. CALCULATE UNITS

FUNCTION: to perform the calculation(s) associated with the unit
(unit operation).

GENERAL FORMAT:

Type 1: calculate units all [[(<arguments>)]] [<if-clause>];

Type 2: calculate units

(<unit>[[(<arguments>)[,<transfer-point>]])

[,<unit>[[(<arguments>)[,<transfer-point>]]]*

[<if-clause>];

GENERAL RULES: <arguments> is a list of integer numbers to be passed to the calculating routine. The first argument is always interpreted as the level of calculation specifying the specific calculation to be made. The interpretation of other arguments depends on the calculating routine. The arguments may represent some input variables or specify the route for the calculating routine or other routines called by that routine.

Transfer-point is the name of the unit where the calculation should start again if convergence has not been achieved.

Other general rules for this command are as follows:

- 1) All units must exist.
- 2) Transfer-point must be the name of a unit specified earlier in the list.
- 3) If level is not specified default level of one will be assumed.
- 4) The number of arguments the user should provide for each unit depends on the unit type. For example, suppose unit type "x" has 2 levels of calculations and minimum and maximum number of arguments have been pre-set in the attached TBS to 2 and 4 respectively. Then the following are the only valid argument lists for calculating a unit of type "x":

(n1,n2)

(n1,n2,n3)

(n1,n2,n3,n4)

Where n1 should be either 1 or 2. The other arguments should be integer numbers between 0 and 99999.

- 5) Before the system passes control to the calculating routine it performs some specific checking for each unit. These are primarily for detecting over- or under-specification. If an error is detected by the system or the calculating routine the execution of the command will be terminated. Moreover, if the command resides in a closed loop the execution of that loop will be terminated.
- 6) If the profile parameter dflag is not equal to one, the system will print a message on the user's terminal whenever it calls a calculating routine.
- 7) If the profile parameter dflag is not equal to two, the system will take over control and prevent the termination of the PEL session, whenever an undetected fatal error occurs in a calculating routine. In this case the execution of the calculate command will be terminated and if the command resides in a closed loop the execution of that loop will also be terminated.

EXAMPLE: calculate u(A,B, C(,A),D(2)); it calculates unit A followed by units B, and C. If calculation in unit C

does not converge, the calculation will be repeated starting with unit A. Once convergence has been achieved in unit C, unit D will be calculated with level equal to two.

II. CALCULATE STREAMS

FUNCTION: To perform specific calculations for streams (e.g., dew point, enthalpy, viscosity, etc.)

GENERAL FORMAT: The same general format as calculate units command.

GENERAL RULES: The same as those for calculate units command.

EXAMPLE: calculate streams (S1,S2,S3);

III. CALCULATE COMPONENTS

FUNCTION: To perform specific calculations for pure components (e.g., vapor pressure, calculating component parameters based on experimental data, etc.).

GENERAL FORMAT: The same general format as calculate units command.

GENERAL RULES: The same as those for calculate units command.

EXAMPLE: calculate components (K2SO4(1));

It is a request for level 1 calculation for the component.

IV. CALCULATE FUNCTIONS

FUNCTION: To compute coefficients of a pre-defined function by correlating raw data, or to perform other calculations for a pre-defined function.

GENERAL FORMAT: The same general format as calculate units command.

GENERAL RULES: The same as those for calculate units command. The calculating routine may prompt the user for some input such as the data to be correlated.

EXAMPLE: calculate function (enthalpy);

CLEAR COMMAND

FUNCTION: To clear the working area and to create a new process.

GENERAL FORMAT: clear;

GENERAL RULES:

- 1) The working area will be cleared. In other words, all the existing units, components, functions, streams, and variables will be deleted.
- 2) The execution of all repeat-loop groups will be terminated (if any).
- 3) A new process will be created; property estimation methods options will be initialized to default options; and the user will be prompted to provide the maximum number of components.
- 4) The opened process file (if any) and the opened component files (if any) will remain opened.
- 5) Profile parameters will remain unchanged.

CLOSE COMMANDS

These commands are used to close a component file or a process file. The close commands can be grouped to the following two commands:

I. CLOSE COMPONENT

FUNCTION: To close a component private or public file.

GENERAL FORMAT: close component [{public|private}] file;

GENERAL RULES:

- 1) If public or private is not specified the default would be the file which is opened. If both files are opened or if both files are closed the default would be public.
- 2) A warning message will be printed if the specified file is already closed.
- 3) If the specified file is opened it will be closed.

EXAMPLE: close c public file;

II. CLOSE PROCESS

FUNCTION: To close a process file.

GENERAL FORMAT: close process file;

GENERAL RULES:

- 1) The opened process file will be closed.
- 2) If no process file is opened, a warning message will be printed.

EXAMPLE: close pr file;

CONNECT COMMANDS

Connect commands may be used to connect units to streams.

I. CONNECT UNIT

FUNCTION: To connect a unit to streams.

GENERAL FORMAT:

Option 1: connect unit <unit> at connection-name=
 <stream> [connection-name=<stream>]*;

Option 2: connect unit <unit> at all = [[,]*<stream>]
 [,]*<stream>*;

GENERAL RULES:

- 1) The unit must exist.
- 2) For option 1 all connection names must belong to the unit type.
- 3) For option 2 repeated commas are allowed to preserve the sequence of connections (inlets, outlets).
- 4) For each unit connection which has already been connected a warning message will be printed.
- 5) For each non-existing stream one will be created. The type of the stream depends on the specified connection. If the specified connection only allows a particular stream type, a stream with that type will be created. Otherwise, a stream with the default type will be created.
- 6) For each connection which does not allow the connection of the specified stream type a warning message will be printed.
- 7) Any attempt to connect an already connected stream will cause a warning message.

EXAMPLE: connect unit a at %c1=S1 %cnct3=S5 feed=S2;

Stream S1 will be connected to the 1st connection, S5 to the 3rd connection and S2 to the connection named feed.

If the feed connection is the 5th connection the following command is equivalent to the above command:
cnct unit a at all=S1,,S5,,S2;

CONTINUE COMMAND

FUNCTION: This is a dummy command which does not perform any action.

GENERAL FORMAT: continue;

COPY COMMANDS

Copy commands may be used to copy data from the public component files into the private component files.

I. COPY COMPONENT

FUNCTION: To copy components data from a public component file into a private component file.

GENERAL FORMAT:

Option 1: copy component {all | (<component-list >) }
[type=type] [override];

Option 2: copy component file;

GENERAL RULES:

- 1) Both the private and public files should be opened.
- 2) For option 1:
 - a) if type is not specified default type will be assumed.
 - b) every component or every specified component of the given

type in the public file will be copied into the private file, provided that the component exists in the public file, and override option is specified. If override option is not specified every component which already exists in the private file will produce a warning message.

- 3) For option 2 all components of all types in the public file will be copied to the private file. The number representing the maximum number of component types in the file will also be copied.

CREATE COMMANDS

Create commands are used to create the process elements. Create commands can be grouped into the following five commands:

I. CREATE UNITS

FUNCTION: To create units.

GENERAL FORMAT: create units (<unit-list>)

[type = unit-type];

GENERAL RULES:

- 1) If type is not specified the default type will be assumed.
- 2) For each non-existing specified unit one will be created.
- 3) For each existing specified unit a warning message will be printed.

EXAMPLE: crt u (Heat1,Heat2) type=Heat_Ex;

II. CREATE STREAMS

FUNCTION: To create streams.

GENERAL FORMAT: create streams (<stream-list>) [type = stream-type];

GENERAL RULES:

- 1) If type is not specified the default type will be assumed.
- 2) For each non-existing specified stream one will be created.
- 3) For each existing specified stream a warning message will be printed.

EXAMPLES: crt s (S1, S2, S3);

crt streams (S4, FEED) type=LIQ_VAP;

III. CREATE COMPONENTS

FUNCTION: To create components.

GENERAL FORMAT: create components (<component-list>)

[type = component-type];

GENERAL RULES:

- 1) If type is not specified the default type will be assumed.
- 2) For each non-existing component in the list, one will be created, as long as the number of existing components does not exceed the pre-specified maximum number of components.
- 3) For each existing specified unit a warning message will be printed.

EXAMPLE: crt c (CaSO₄, H₂O, CaS, H₂);

IV. CREATE FLOW

FUNCTION: To create the flow of components in a stream. In other words, to specify the components that are flowing in a stream.

GENERAL FORMAT:

```
create flow {all|(<stream-list>)}
           {all|(<component-list>)};
```

GENERAL RULES:

- 1) For each non-existing stream or component, a warning message will be printed.
- 2) For each existing stream that does not permit the flow of a specified component type a warning message will be printed.
- 3) For each component which already exists in a given stream a warning message is produced.

EXAMPLE: crt flow (S₁, S₂) all;

All the existing components will be present in streams S₁, and S₂.

V. CREATE FUNCTIONS

FUNCTION: To create a pre-defined function or to define and create an analytical function.

GENERAL FORMAT:

Type 1: To create a pre-defined function:

```
create function (<function-list>)
[ type = function-type];
```

Type 2: To create a user defined function:

```
create function
(<function>(<dummy-argument-list>) =<expression>
[,<function>(<dummy-argument-list>) =<expression> ]*);
```


GENERAL RULES:

- 1) For type 1:
 - a) If type is not specified, the default type will be assumed.
 - b) For each existing function a warning message will be printed.
 - c) For each non-existing function one with the specified type will be created.
- 2) For type 2:
 - a) None of the functions must exist.
 - b) The expressions in addition to numbers and dummy arguments could also refer to other functions and variables. It should be noted that the other variables referred to, would be substituted with their values at that time and will remain unchanged.

EXAMPLES: crt functions (H,S,CP) type=L2;
 crt function (xxx(x,y)=x/y+h(s(cp(y))));

DELETE COMMANDS

Delete commands are used to delete an object from the working area.
 Delete commands can be grouped into the five following commands:

I. DELETE UNITS

FUNCTION: To remove a unit data structure.

GENERAL FORMAT: delete units {all|(<unit-list>)};

GENERAL RULES:

- 1) For each existing specified unit the corresponding source and destination of all inlet and outlet streams will be initialized, and unit will be deleted.
- 2) For each non-existing unit a warning message will be printed.

EXAMPLE: delete units all; \$all units will be erased \$

II. DELETE STREAMS

FUNCTION: To remove a stream data structure.

GENERAL FORMAT: delete streams {all|(<stream-list>)};

GENERAL RULES:

- 1) For each existing specified stream its unit connections will be initialized and the stream will be erased.
- 2) For each non-existing stream a warning message will be printed.

EXAMPLE: delete s (S2, FEED);

III. DELETE COMPONENTS

FUNCTION: To remove a component data structure.

GENERAL FORMAT: delete components {all|(<component-list>)};

GENERAL RULES:

- 1) For each non-existing component in the list, a warning message will be printed.
- 2) For each component that is present in any stream a warning message will be printed.

- 3) Only those existing components which are not present in any stream at that time, will be erased.

IV. DELETE FLOW

FUNCTION: To delete the flow information portion of streams. In other words to indicate that some components are not longer present in a stream.

GENERAL FORMAT: delete flow {all|(<stream-list>)}
 {all|(<component-list>)};

GENERAL RULES:

- 1) For each non-existing stream or component a warning message will be printed.
- 2) For each specified existing component which is not present in a specified existing streams a warning message will be printed.
- 3) Flow information of all specified existing components in all specified existing streams which contain those components will be erased.

EXAMPLE: del f all all ;

The flow information of all components in all streams will be deleted. In other words, there would be no component in any stream.

V. DELETE FUNCTIONS

FUNCTION: To delete user-created functions.

GENERAL FORMAT: delete functions {all|(<function-list>)};

GENERAL RULES:

- 1) For each non-existing function a warning message will be printed.
- 2) A function cannot be deleted if it is referred by an existing user-defined function. Therefore, for each such function a warning message will be printed.

EXAMPLE: delete functions (A,B,C);

DELETEF COMMANDS

Deletef commands are used to delete components in a component file, or to delete processes in a process file. Deletef commands can be grouped into the following two commands.

I. DELETEF COMPONENTS

FUNCTION: to delete components in a component private file.

GENERAL FORMAT: deletef components {all|(<component-list>)
[type=type];

GENERAL RULES:

- 1) If type is not specified the default type will be assumed.
- 2) The component private file should be opened.
- 3) The user should have the proper access to the file.
- 4) Each specified component which does not exist in the file will produce a warning message.
- 5) Each specified component which exists in the file will be deleted.

EXAMPLE: deletef components (C1,C2) type=one;

II. DELETEF PROCESS

FUNCTION: To delete specified processes in a process file.

GENERAL FORMAT: deletef process {all|(<process-list>)};

GENERAL RULES:

- 1) The process file should be opened.
- 2) The user should have the proper access to the file.
- 3) Each specified process which does not exist in the file will raise a warning message.
- 4) Each specified process which exists in the file will be deleted.

EXAMPLE: deletef pr (E1,E2);

DISCONNECT COMMANDS

Disconnect commands are used to disconnect units from streams.

I. DISCONNECT UNIT

FUNCTION: To disconnect a unit from streams.

GENERAL FORMAT: disconnect unit <unit> at {all|(<connection-list>)};

GENERAL RULES:

- 1) The unit must exist.
- 2) If all option is specified, the unit will be disconnected at all connections.
- 3) For every specified connection, the unit will be disconnected from the stream connected to that connection. If the unit is already disconnected at that connection a warning message will be printed.

EXAMPLE: disconnect unit MHD_Channel at (feed);

END COMMAND

FUNCTION: To terminate the session.

GENERAL FORMAT: end ;

GENERAL RULES:

- 1) The user is reminded if he or she has created some output in his output file (i.e. output.GPES segment) during the session.
- 2) The working area will be cleared. Therefore the user should save his current process before issuing this command, if that process is of any interest to him.
- 3) The executive session will be terminated.

ESCAPE COMMAND

FUNCTION: To transfer control to the Multics command level where the user may execute any number of Multics commands or invoke other programs.

GENERAL FORMAT: escape;

GENERAL RULES:

- 1) To return to PEL command level, the user should enter a blank line.

HELP COMMANDS

FUNCTION: To print a description of a PEL command.

GENERAL FORMAT:

Type 1: help commands;

- Type 2: help notation;
- Type 3: help <object>;
- Type 4: help <verb><object>;
- Type 5: help <verb>;
- Type 6: help all;

GENERAL RULES:

- 1) For type 1: all the PEL commands will be tabulated.
- 2) For type 2: syntax notation conventions used in describing the general format of commands will be printed.
- 3) For type 3: A description of the given object will be given.
- 4) For type 4: A description of the given command will be printed. The description includes the function, general format, general rules, and one or more more examples.
- 5) For type 5: A description of the given group of commands will be given. Each command in that group will be described as mentioned for type 4.
- 6) For type 6: The printout includes the following:
 - a) Table of PEL commands (type 1);
 - b) Syntax notation conventions (type 2);
 - c) Description of each of the seven command objects (type 3 for each object);
 - d) Description of all the PEL commands (type 5 for each command).

EXAMPLE: help commands; \$table 7.1 is the output of this command.\$

INCLUDE COMMANDS

Include commands are used to add some components to a component file.

I. INCLUDE COMPONENT

FUNCTION: To include the specified components whose parameters are provided by the user from the terminal, into a private component file.

GENERAL FORMAT: include component (<component-list>)[type=type]
[override];

GENERAL RULES:

- 1) If type is not specified, the default type will be assumed.
- 2) The component private file should be opened.
- 3) The user should have the proper access on the file.
- 4) For each component the user will be prompted to provide the value of each parameter. If the component is already present in the file and override option is not provided a warning message will be printed.

LEAVE COMMAND

FUNCTION: To leave the current attached TBS and attach another one.

GENERAL FORMAT: leave [brief];

GENERAL RULES:

- 1) The working area will be cleared. The current process will be lost if the user does not save it before issuing this command.

- 2) The opened component files (if any) will be closed.
- 3) The user will be asked to provide the name of the new TBS. The new TBS will be attached. If brief option is not provided some information about the new TBS will also be printed.
- 4) All profile parameters will be initialized to the TBS default parameters.
- 5) A new process will be created; property estimation methods option will be initialized to the TBS default options; and the user will be prompted to provide the maximum number of components.
- 6) The opened process files (if any) will remain opened.

EXAMPLE: leave bf;

A.5.9 LET COMMANDS

Let commands are used to duplicate or copy an object. The let commands can be grouped into the following five commands:

I. LET UNIT

FUNCTION: To duplicate or copy a unit.

GENERAL FORMAT: let unit <unit>=<unit>;

GENERAL RULES:

- 1) The unit on the righthand side must exist.
- 2) If the unit on the lefthand side exists, it must be same type as the unit on the righthand side.

- 3) If the unit on the lefthand side (target unit) does not exist, one with the same type as the other unit will be created.
- 4) The values and value types of all parameters of the unit on the righthand side will be assigned to the parameters of the target unit.
- 5) The inlet and outlet connections remain unchanged.

EXAMPLE: let unit A=B;

Suppose unit B has four parameters as follows:

p1 "unspecified"
 p2 "specified" 10
 p3 "assumed" 20
 p4 "calculated" 2

Unit A parameters would be:

p1 "unspecified"
 p2 "specified" 10
 p3 "assumed" 20
 p4 "calculated" 2

II. LET STREAM

FUNCTION: To duplicate or copy a stream;

GENERAL FORMAT: let stream <stream>=<stream>;

GENERAL RULES:

- 1) The stream on the righthand side must exist.

- 2) If the target stream exists, it must be the same as the other stream.
- 3) If the target stream does not exist one with the same type as the other stream will be created.
- 4) All the phase parameters of each phase of the stream on the righthand side will be assigned to those of the target stream. Flow information and flow parameters are not copied.
- 5) The source and the destination of each stream remain unchanged.

EXAMPLE: let stream S1 = S2;

III. LET COMPONENT

FUNCTION: To copy or duplicate a component.

GENERAL FORMAT: let component <component>=<component>;

GENERAL RULES:

- 1) The component on the righthand side must exist.
- 2) If the component on the lefthand side exists, it must be the same as the other component.
- 3) If the target component does not exist one with the same type as the other component will be created, provided that the number of components does not exceed the pre-specified maximum number of components.
- 4) All parameters of the component on the righthand side will be assigned to those of the target component.

EXAMPLE: let c IC4H10 = NC4H10;

IV. LET FLOW

FUNCTION: To duplicate or copy the flow information of a stream.

GENERAL FORMAT: let flow<stream> = <stream>;

GENERAL RULES:

- 1) The stream on the righthand side must exist.
- 2) If the stream on the lefthand side exists, it must be same as the other stream.
- 3) If the stream on the lefthand side does not exist one with the same type as the other stream will be created.
- 4) The flow of all components in all phases will be copied from righthand side stream to the target stream. In other words only those components will be present in the lefthand side stream that are present in the righthand side stream, and all flow parameters will be copied.
- 5) All phase parameters of the target stream remain unchanged.
- 6) The source and destination of the target stream remain unchanged.

EXAMPLE: let flow S1 = S2;

V. LET FUNCTION

FUNCTION: To duplicate or copy a pre-defined function.

GENERAL FORMAT: let function <function> = <function>;

GENERAL RULES:

- 1) The function on the righthand side must exist and must be a pre-defined type.
- 2) If the function on the lefthand side (target function) exists, it must be same type as the function on the righthand side.
- 3) If the target function does not exist, one with the same type as the other function will be created.
- 4) All parameters (coefficients) of the function on the righthand side will be assigned to the parameters of the target function.

EXAMPLE: let fn A = B;

LETA COMMANDS

These commands function exactly like the let commands with one exception. In let commands both value and value type of each parameter will be copied, whereas in leta commands only the value of each parameter will be copied. The value type of all target parameters which are not unspecified will be set to "assumed".

The general format and rules of these commands are the same as the let commands (replace let with leta). The relationship between let and leta commands is the same as the relationship between specify and assume commands, or between read and reada commands.

LIST COMMANDS

List commands are used to list the process elements in the working

area. List commands can be grouped to the following five commands.

I. LIST UNITS

FUNCTION: To list the units in the process.

GENERAL FORMAT: list units;

GENERAL RULES: 1) The name of each unit in the process will be printed.

II. LIST STREAMS

FUNCTION: To list the streams in the process.

GENERAL FORMAT: list streams;

GENERAL RULES: 1) The name of each stream in the process will be printed.

III. LIST COMPONENTS

FUNCTION: To list the components in the process.

GENERAL FORMAT: list components;

GENERAL RULES: 1) The name of each component in the process will be printed.

IV. LIST FUNCTIONS

FUNCTION: To list the functions in the process.

GENERAL FORMAT: list functions;

GENERAL RULES: 1) The name of each function in the process will be printed.

V. LIST VARIABLES

FUNCTION: To list the simple variables.

GENERAL FORMAT: list variables;

GENERAL RULES: 1) The name and value of each simple variable in the process will be printed.

LISTF COMMANDS

listf commands are used to list components or processes in the component files or process files. listf commands can be grouped into the following two commands:

LISTF COMPONENTS

FUNCTION: To list all the components in a component file.

GENERAL FORMAT: listf component {all|[type=type]} [{private|public}];

GENERAL RULES: 1) If private or public is not specified the default would be the file which is opened. If both files are opened or if both files are closed, the default would be the public file.

2) The specified file should be opened.

3) If "all" option is given, all components of every type will be listed.

4) If neither type nor all is specified the default type will be used.

5) If the all option is not given the components of the given type or default type will be listed.

EXAMPLE: listf c private; components of default type in the opened private component file will be listed.

II. LISTF PROCESS

FUNCTION: To list the processes in a process file.

GENERAL FORMAT: listf process;

GENERAL RULES: 1) The process file should be opened.

EXAMPLE: lf pr;

LISTT COMMANDS

listt commands are used to list the types of objects in the attached TBS. listt commands can be grouped into the following commands:

I. LISTT UNIT

FUNCTION: To list the unit types (templates).

GENERAL FORMAT: listt units;

GENERAL RULES: 1) All unit types will be listed.

II. LISTT STREAMS

FUNCTION: To list the stream types (templates).

GENERAL FORMAT: listt streams;

GENERAL RULES: 1) All stream types will be listed.

III. LISTT COMPONENTS

FUNCTION: To list the component types (templates).

GENERAL FORMAT: listt components;

GENERAL RULES: 1) All component types will be listed.

IV. LISTT FUNCTIONS

FUNCTION: To list the pre-defined function types (templates).

GENERAL FORMAT: listt functions;

LOAD COMMANDS

Load commands are used to retrieve components from a component file or a process from a process file into the working area. Load commands can be grouped into the following two commands:

I. LOAD COMPONENTS

FUNCTION: To copy the specified components from a public or private component file into the working area.

GENERAL FORMAT: load component (<component-list>) [type=type]
 [{public|private}];

- GENERAL RULES:
- 1) If type is not specified the default component type would be assumed.
 - 2) If public or private is not specified the default would be the file which is opened. If both files are opened or if both files are closed the default would be the public file.
 - 3) The specified or implied file should be opened.
 - 4) For every specified component which already exists in the working area a warning message will be printed.
 - 5) For every specified component which does not exist in the specified file, a warning message will be printed.
 - 6) When a component is loaded the value types of all its parameters will be set to be specified.

EXAMPLE: load c(CO2,CO);

II. LOAD PROCESS

FUNCTION: To load a process from a process file into the working area.

GENERAL FORMAT: load process <process>;

- GENERAL RULES:
- 1) The process file should be opened.
 - 2) The specified process must exist in the process file.
 - 3) If the process is not compatible with the current generation of the system or attached TBS the user will be given the option of continuing or ignorng the command.
 - 4) All active repeat commands (if any) will be terminated.
 - 5) The working area will be cleared. Therefore if current process is of any interest to the user, he should save it, before issuing this command.
 - 6) The specified process will be loaded into the working area.
 - 7) The property estimation methods in effect and maximum number of components will be those of the loaded process.

EXAMPLE: load pr test;

LOOP COMMAND

FUNCTION: To designate the end of a group of commands headed by a repeat command to be repeated.

GENERAL FORMAT: loop;

- GENERAL RULES:
- 1) There must be an unterminated repeat command preceding the loop command.

NEWS COMMAND

FUNCTION: To print the news of the system and the attached TBS.

GENERAL FORMAT: news;

OPEN COMMAND

Open commands are used to open a component or process file or to create such a file. Open commands can be grouped into the two following commands:

I. OPEN COMPONENT

FUNCTION: To open a component file.

GENERAL FORMAT: open component [{public|private}] file (<file>);

- GENERAL RULES:
- 1) If public or private is not specified, default would be the file which is not opened. If both files are closed or if both files are opened the default would be public file.
 - 2) The user will be prompted to provide the pathname of the file.
 - 3) If the request is for a public file, the specified public file should exist.
 - 4) The user should have the proper access, for opening the file.
 - 5) If the request is for a private file, and if the specified file does not exist a new component file will be created provided that:
 - a) The user has the proper access for creating a new file under the given pathname, and

- b) The user is willing to create a new one.
- 6) If the file is not compatible with the attached TBS the user will be given the option of continuing the opening or ignoring the command.
- 7) Public files are in fact the private file of TBS system administrators. They are created and updated by the TBS system administrators.
- 8) Once the user has opened a private (or public) file it remains open until:
 - a) He closes the private (or public) file,
 - b) He opens another private (or public) file,
 - c) He terminates the file,
 - d) He leaves the TBS, or
 - e) He terminates the session.

EXAMPLE: open c file (A);

II. OPEN PROCESS

FUNCTION: To open or create a new process file.

GENERAL FORMAT: open process file (<file>);

- GENERAL RULES:
- 1) The user will be prompted to provide the pathname of the file.
 - 2) If the file exists the user should have the proper access for opening of the process file.
 - 3) If the file does not exist a new one will be created provided that:

- a) The user has the proper access for creating a new file under the given pathname, and
 - b) The user is willing to do so.
- 4) If the file is incompatible with the attached TBS the user will be given the option of continuing the opening or ignoring the command.
- 5) Once a process file is opened it remains opened until:
- a) The user closes the file
 - b) The user opens another file
 - c) The user terminates the file, or
 - d) The user terminates the session.

EXAMPLE: open pr file (x);

PRINT COMMAND

Print commands are used to print some information about a process element. These commands can be grouped into the following seven commands:

I. PRINT UNITS

FUNCTION: To print some information about units.

GENERAL FORMAT: print units {all|(<unit-list>)} {unspecified|assumed|specified|calculated|parameters|connections|type|all}[<if-clause>];

GENERAL RULES: 1) All specified units must exist.

2) Print option indicates the type of information to be printed about each specified unit or every unit:

- a) unspecified: unspecified unit parameters will be printed.
- b) assumed: assumed unit parameters will be printed.
- c) specified: specified unit parameters will be printed.
- d) calculated: calculated unit parameters will be printed.
- e) parameters: all the unit parameters will be printed.
- f) connections: inlet and outlet streams will be listed.
- g) type: unit type will be printed.
- h) all: type, connections, and all the unit parameters will be printed.

EXAMPLE: `print u (A) unspecified;` Unspecified parameters of unit "A" will be printed.

II. PRINT STREAMS

FUNCTION: To print some information about streams.

GENERAL FORMAT: `print streams {all| (<stream-list>)}`

`{unspecified|assumed|specified|calculated|parameters|`

connections|type|all} [<if-clause>;

GENERAL RULES: 1) All specified streams must exist.
2) Print option indicates the type of information to be printed about each specified stream or about every stream:

- a) unspecified: unspecified phase (stream) parameters will be printed.
- b) assumed: assumed phase (stream) parameters will be printed.
- c) specified: specified phase (stream) parameters will be printed.
- d) calculated: calculated phase (stream) parameters will be printed.
- e) parameters: all the phase (stream) parameters will be printed.
- f) connections: source and destination of the stream will be printed.
- g) type: stream type will be printed.
- h) all: type, connections, and all the phase (stream) parameters will be printed.

FUNCTION: print s all connections; Source and destination of each stream will be printed.

III PRINT COMPONENTS

FUNCTION: To print specific information about components.

GENERAL FORMAT: print components{all|(<component-list>)}
 {unspecified|assumed|specified|calculated|parameters|
 streams|type|all} [<if-clause>];

GENERAL RULES:

- 1) All specified components must exist.
- 2) The given print option indicates the specific information to be printed about each specified component or each component.
 - a) unspecified: unspecified component parameters will be printed.
 - b) assumed: assumed component parameters will be printed.
 - c) specified: specified component parameters will be printed.
 - d) calculated: calculated component parameters will be printed.
 - e) parameters: all the component parameters will be printed.
 - f) streams: streams containing the component will be listed.
 - g) type: component type will be printed.
 - h) all: type, parameters and streams will be printed.

EXAMPLE: print c (CO2) streams if (A>B);

If variable A is greater than variable B the streams
 containing
 component CO2 will be listed.

IV PRINT FLOW

FUNCTION: To print specific information about components flowing in streams.

GENERAL FORMAT: print flow{all|(<stream-list>)}
 {unspecified|assumed|specified|calculated|parameters|
 components|all} [<if-clause>];

GENERAL RULES:

- 1) All specified streams must exist.
- 2) The given print option indicates the type of information to be printed about each specified stream or each stream:
 - a) unspecified: unspecified flow parameters will be printed.
 - b) assumed: assumed flow parameters will be printed.
 - c) specified: specified flow parameters will be printed.
 - d) calculated: calculated flow parameters will be printed.
 - e) parameters: all the flow parameters will be printed.
 - f) components: components present in the stream will be listed.
 - g) all: parameters and components will be printed.

EXAMPLE: p f all c; \$components present in each stream will be listed.\$

V PRINT FUNCTIONS

FUNCTION: To print specific information about pre-defined functions.

GENERAL FORMAT: print functions {all|(<function-list>)}
 {unspecified|assumed|specified|calculated|parameters|
 functions|type|all} [<if-clause>];

GENERAL RULES:

- 1) All specified functions must exist and be of the pre-defined type.
- 2) The given print option indicates the specific information to be printed about each specified function or each function.

- a) unspecified: unspecified parameters will be printed.
- b) assumed: assumed parameters will be printed.
- c) specified: specified parameters will be printed.
- d) calculated: calculated parameters will be printed.
- e) parameters: all the function parameters will be printed.
- f) functions: the number of times the existing user-defined functions referring to the function will be printed.
- g) type: the function type will be printed.
- h) all: type, parameters, and functions will be printed.

EXAMPLE: fn (H) calculated;

\$calculated parameters of function H will be printed.\$

VI PRINT VARIABLES

FUNCTION: To print values of variables.

GENERAL FORMAT: print variables {all|(<variables-list>)}
[<if-clause>];

GENERAL RULES:

- 1) Variables can be simple or qualified.
- 2) If all option is used all simple variables will be printed.

EXAMPLE: print v (u.distil.t,x); value of parameter "t" of unit distil and value of variable x will be printed.

VII PRINT PROCESS

FUNCTION: To print information about the entire process.

GENERAL FORMAT: print process {flowsheet|all} [<if-clause>];

GENERAL RULES:

- 1) If flowsheet option is specified a flowsheet diagram of the process will be printed (it is not implemented yet).
- 2) If all option is specified the following information will be printed:

- a) Process flowsheet (not implemented yet).
- b) A list of all units.
- c) A list of all streams.
- d) A list of all components.
- e) A list of all user-defined functions.
- f) A list of all pre-defined functions.
- g) A list of all simple variables.
- h) All information about each unit.
- i) All information about each stream.
- j) All information about each component.
- k) All information about each pre-defined function.

EXAMPLE: `print pr all if (u.test.flag=1);` If parameter flag of unit test be equal to one, all information about the entire process will be printed.

PRINTF COMMANDS

Printf commands are used to print information about components on a component file.

I. PRINTF COMPONENTS

FUNCTION: To print parameters of components on a component file.

GENERAL FORMAT: `printf components {all|(<component-list>)}[type=type]
[{public|private}];`

GENERAL RULES: 1) If type is not specified default component type will be assumed.

2) If public or private is not specified the default would be the file which is opened. If both files are opened or if both files are closed the default would be public.

- 3) The specified file should be opened.
- 4) The parameters of each specified component which exists on the file will be printed.
- 5) For each specified component which does not exist on the file, a warning message will be printed.

EXAMPLE: `printf c (CO) public;` Parameters of component CO of default type on the opened public file will be printed.

PRINTT COMMANDS

FUNCTION: To print information about the templates of the attached TBS.

GENERAL FORMAT: Type 1: `printt {unit|stream|component|function} [[type=type]|all];`

Type 2: `printt dimension {<integer-number>}|all];`

Type 3: `printt property {<integer-number>}|all];`

Type 4: `printt ctlinfo;`

Type 5: `printt vsctable;`

Type 6: `printt all;`

- GENERAL RULES:
- 1) For Type 1, if type is not specified the default type for the given object will be assumed.
 - 2) For Type 1, the template of the specified type or every type of the given object will be printed.
 - 3) For Type 2, the template of the given dimension type or the entire dimension table will be printed.
 - 4) For Type 3, the template of property estimation methods of the given property or all properties will be printed.

- 5) For Type 4, the template for the control information of the attached TBS will be printed.
- 6) For Type 5, the value status codes table will be printed.
- 7) For Type 6, all templates will be printed. The output may be used as the Users Reference Manual of the attached TBS.

EXAMPLES: printt component;

The template of the default component type will be printed.

Printt dimension 1;

The template for the dimension type 1 will be printed.

PROFILE COMMAND

FUNCTION: To allow the user to provide his/her desired profile parameters.

GENERAL FORMAT:

Type 1: profile print;

Type 2: profile [{sdigit|ddigit|dflag|output|input}=
<integer-number>]*;

Type 3: profile default;

GENERAL RULES:

- 1) The profile parameters are as follows:

sdigit: number of significant digits to be used in response to print commands. It should be between 1 and 14 (inclusive).

ddigit: Number of decimal digits to be used in response to print commands it should be between 0 and sdigit (inclusive).

output: It must be either 0 or 1, the value of zero indicates that the output commands should be printed on user's terminal, the value of one indicates that the output of output commands should be saved on a segment which later on could be printed on a high speed printer or to be kept for any other purposes. Upon specifying the value of one to this profile parameter, the system will create a segment called "output.GPES" in the user's current working directory, if one does not already exist. The output saved on this segment in one session follows the previously created outputs and is preceded by a headline specifying the date on which the output was created.

dflag: Debugging flag is a number between zero and three indicating whether additional information regarding command processing to be printed. dflag of zero indicates that:

- a) whenever the system calls upon a calculating routine a message be printed on the terminal, and
- b) if any fatal error occurs in the calculating routine, the GPES executive take over control, and thus prevent the termination of the session. The system should ignore the command for which the routine was invoked, and accept a new command.

dflag of one is a request for (b) only. dflag of two is a request for (a) only. dflag of three is a request for both (a) and (b) and the printout of some additional information regarding the internal operation of the system. dflag of three is recommended for use by the GPES administrator for debugging the system. dflag of two is recommended for use by TBS administrators and TBS programmers for debugging TBS programs. dflag of zero or one is recommended for normal operation. The default value of dflag is either zero or one and is set by the TBS administrator.

input: Currently it must be 0. It indicates that the system should receive the commands and other inputs from the terminal.

- 2) For Type 1: The current value of profile parameter will be printed.
- 3) For Type 2:
 - a) The sequence of listing parameters is immaterial.
 - b) Any number of parameters may be specified.
 - c) The system will override the already specified or default options. The specified parameters will be in effect until they are overridden by another profile command or initialized to the default profile parameters of a new TBS (when attaching a new TBS).

4) For Type 3: All profile parameters will be initialized to the default values of the current attached TBS. They will remain in effect until they are overridden by another profile command or initialized to the default parameters of a new TBS (when attaching a new TBS).

5) When attaching a TBS the profile parameters will be set to the default profile parameters of the TBS. The default parameters are:

sdigit = between 0 and 14 (inclusive) set by the TBS system administrator.

ddigit = between 0 and sdigit (inclusive) set by the TBS system administrator.

dflag = 0 or 1 set by the TBS system administrator.

output = 0

input = 0

EXAMPLES:

**COMMAND: prof sdigit = 10 ddigit = 2;

**COMMAND: prof print;

sdigit = 10

ddigit = 2

dflag = 0

output = 0

input = 0

**COMMAND: prof default;

**COMMAND: prof print;

sdigit = 14

ddigit = 13

dflag = 0

output = 0

input = 0

READ COMMANDS

Read commands are used to instruct the system to accept data from the terminal and assign it to parameters or variables. The user is prompted to provide data for each specified parameter or variable. The user's response should be in the following form:

```
[[{+|-}]<number> ['dimension']]
```

The dimension should be appropriate for the corresponding parameter. If dimension is not specified the default is the standard units for that parameter. If no value is provided (a blank line is entered) for a variable or parameter the value of that parameter or variable remains unchanged. The read commands can be grouped to the following six commands:

I. READ UNIT

FUNCTION: To accept data from the terminal and assign it to unit parameters.

GENERAL FORMAT:

```
Type 1: read unit (<unit-list>) {all|(<parameter-list>)}
        [<if-clause>];
```

```
Type 2: read unit all [<if-clause>];
```

GENERAL RULES:

- 1) For type 1:
 - a) Each unit in the list must exist.
 - b) All units in the list must be the same type.
 - c) All specified parameters must belong to the unit type.
 - d) The user is prompted to provide data for every specified parameter for each specified unit.

- 2) For Type 2: The user is prompted to provide data for every parameter of each existing unit.

EXAMPLE:

```

**COMMAND: read u(u1) (t,p,n);

**enter the following unit parameters:

unit u1

t: 100 'degr'

p: 100 'dum'

***ERROR*** i 21 'dum' is an invalid dimension for the specified
        parameter.

**reenter the input: 100 'atm'

        n: $parameter n remains unchanged$

**COMMAND:

```

II. READ STREAMS

FUNCTION: To accept data from the terminal and assign it to stream parameters.

GENERAL FORMAT:

```

Type 1: read stream (<stream-list>)
        [<phase-field>]{all|(<parameter-list>)}
        [[<phase-field>]{all|(<parameter-list>)}]*
        [<if-clause>];

```

```

Type 2: read stream all [<if-clause>];

```

GENERAL RULES:

- 1) Each stream in the list must exist.
- 2) Each stream in the list must be the same type.
- 3) If phase-field is not specified phase 0 (total stream) will be assumed.

- 4) Each specified parameter should be a valid phase parameter of the given phase of the stream type.
- 5) The user is prompted to provide data for every specified phase (stream) parameter for each specified stream.
- 6) For Type 2: The user is prompted to provide data for every phase (stream) parameter for each stream.

EXAMPLE: rd stream (feed) (wc,t);

III. READ COMPONENT

FUNCTION: To accept data from the terminal and assign it to component parameters.

GENERAL FORMAT:

Type 1: read component (<component-list>)
 {all|(<parameter-list>)}
 [<if-clause>;

Type 2: read component all [<if-clause>;

GENERAL RULES:

- 1) Each component in the list must exist.
- 2) All components in the list must be the same type.
- 3) All specified parameters must belong to the component type.
- 4) The user will be prompted to provide data for every specified parameter for each specified component.
- 5) For Type 2, the user is prompted to provide data for every parameter of each existing component.

EXAMPLE:

read c (c1) (t) if(-1);

IV. READ FLOW

FUNCTION: To accept data from the terminal and assign it to flow parameters.

GENERAL FORMAT:

```
Type 1: read flow (<stream-list>)
          [<phase-field>]{all|(<flow-parameter-list>)}
          [[<phase-field>]{all|(<flow-parameter-list>)}]*
          [<if-clause>];
```

```
Type 2: read flow all [<if-clause>];
```

GENERAL RULES:

- 1) Each specified stream should exist.
- 2) All specified streams should be the same type.
- 3) If phase-field is not specified the default would be phase 0 (total stream).
- 4) Every specified parameter should be a valid flow parameter of the given phase of the stream type.
- 5) The user is prompted to provide data for every specified flow parameter for each stream.
- 6) For Type 2 user will be prompted to provide data for every flow parameter of each existing stream.

EXAMPLE:

```
read flow (feed) all; $read all flow parameters of all
components in phase 0 of stream feed$
read flow (feed)(CO(f),(CO2(f))); $read flow parameter "f" of
components CO and CO2 in phase 0 of stream feed$.
```

V. READ FUNCTION

FUNCTION: To accept data from the terminal and assign to pre-defined function parameters.

GENERAL FORMAT: Type 1: read function (<function-list>)
 {all|(<parameter-list>)} [<if-clause>];
 Type 2: read function all [<if-clause>];

GENERAL RULES:

- 1) Each function in the list must exist.
- 2) All functions in the list must be pre-defined and the same type.
- 3) All specified parameters must belong to the function type.
- 4) The user is prompted to provide data for every specified parameter for each specified function.
- 5) For Type 2, the user is prompted to provide data for every parameter of each existing function.

EXAMPLE: read fn all if(A>B); If variable A is greater than variable B, the user will be prompted to provide data for every parameter of each existing pre-defined function.

VI. READ VARIABLES

FUNCTION: To accept data from the terminal and assign it to variables.

GENERAL FORMAT:

Type 1: read variables (<variable-list>)
 [<if-clause>];
 Type 2: read variables all [<if-clause>];

GENERAL RULES:

- 1) Variables can be simple or qualified.
- 2) The user will be prompted to provide data for each specified variable.
- 3) For Type 2, the user will be prompted to provide data for each existing simple variable.

EXAMPLE:

```
read v(u.a.t,x,y);
```

READA COMMANDS

These commands function exactly like the read commands with one exception. A read command functions as though the input data has been assigned to parameters by a specify command, while a reada command functions as though the input data has been assigned to parameters by an assume command. In other words the relationship between read and reada commands are the same as the relationship between specify and assume command or between let and leta commands. The general rules and formats of reada commands are the same as the read commands (replace read with reada). For further explanation refer to read commands.

NOTE: For simple variables read and reada commands function exactly the same.

REPEAT COMMAND

FUNCTION: It heads a group of commands and specifies repetitive execution of the commands within the group.

GENERAL FORMAT: Type 1: repeat for <variable>
 from (<expression>['dimension'])
 to (<expression>['dimension'])

```
[by(<expression>['dimension'])]
```

```
[while(<expression>)];
```

Type 2: repeat for

```
<variable>=<expression>['dimension']
```

```
[,<expression>[dimension]]*)
```

```
[while(<expression>)];
```

GENERAL RULES:

- 1) For type 1, if the BY option is not specified the value of one will be assumed.
- 2) The variable (control-variable) can be simple or qualified.
- 3) For qualified variables valid dimensions may be provided.
- 4) loop command designates the end of the sequence of commands to be repeated.
- 5) Each expression in the command should be evaluable before the control-variable has been assigned. For example, if u.a.t is unspecified the following command is in error: repeat for u.a.t from(10) to (10*u.a.t);
- 6) The WHILE clause specifies that before each repetition of the command sequence, the associated expression be evaluated. If it is positive or zero the commands within the group be executed, otherwise, the execution of the group be terminated.
- 7) For type 1, the FROM (expression) represents the initial value of the control-variable. The BY (expression) represents the increment to be added to the control-variable after each execution of the commands in the

group. The TO (expression) represents the terminating value of the control variable. Execution of the commands in the group terminates as soon as:

- a) the WHILE (expression) (if any) becomes negative,
- b) the value of the control-variable, when updated is greater than the TO (expression), if the BY (expression) is positive or zero, or
- c) the value of the control-variable, when updated, is less than the TO (expression), if the BY (expression) is negative.

8) For type 2, initially the value of the first expression will be assigned to the control-variable. In the 2nd repetition the value of the 2nd expression will be assigned to the control-variable and in the 3rd repetition the value of 3rd expression and so on. The execution of the commands in the group terminates as soon as:

- a) the WHILE expression (if any) becomes negative, or
- b) the list of expressions to be assigned to the control-variable is exhausted.

9) PEL commands can be classified to the following groups with respect to their appearance inside a repeat-loop group.

- a) Commands that are not allowed inside a repeat-loop group. The following commands are not allowed inside a repeat-loop group:

delete

let

leta

unspecify

b) Commands that terminate the group

The following commands when appearing after a repeat command will terminate the execution of the sequence of commands to be repeated:

clear

end

leave

load process

stop

The first four commands will terminate the most outside group. In other words, they terminate each existing group. The stop command terminates only the corresponding group.

c) Non-Repetitive Commands

The following commands when appearing inside the group will be executed only once regardless of the condition of the group:

bugs

listt

close

load component

connect

news

continue

open

copy

printf

create

printt

deletef

profile

disconnect

save

escape terminate
help use
include
list
listf

d) Repetitive Commands

Only the following commands when appearing inside the group can be executed, any number of times, depending on the condition of the group:

assume
calculate
loop
print
read
reada
repeat
specify

- 10) A repeat command is called unexecutable when after the control-variable has been initialized (the first iteration), the WHILE expression is negative; or for type 1 the control variable is greater than the TO expression and the BY expression is positive or zero; or the control variable is less than the TO expression and the BY expression is negative. If a repeat command is unexecutable, non-repetitive commands that may follow this command will be executed only once as usual. Repetitive commands that may follow the command will be

checked for syntax, but will not be executed, as long as, the condition in the group does not permit. If the repeat command is nested inside another repeat-loop group, conditions may be relaxed and the execution of the group may be resumed. But if the repeat command is external (the most outside group), the holding conditions will remain unchanged and the corresponding loop command that will follow will terminate the group without achieving any purpose from the group. Hence, external unexecutable repeat commands will produce warning messages.

- 11) If a repeat command appears in a group headed by an unexecutable repeat command, the repeat command itself will be executed, in other words, its control-variable will be initialized to the specified value, but the repetitive commands (except the repeat commands) following it will not be executed.
- 12) Although the same variable can be used as the control-variable of several nested repeat-loop groups, if proper care is not exercised it may result in an endless loop.
- 13) Although the user can change the value of the control variable, FROM, TO, and BY expressions inside a repeat-loop group, if proper care is not taken it may result in an endless loop.

EXAMPLES: repeat for s.feed.s.t from (10) to (100) by (10);

.
.
.

```
repeat for seed = (exp1, exp2, exp3) while (air<1.3);
```

```
.
```

```
.
```

```
.
```

```
loop;
```

```
.
```

```
.
```

```
.
```

```
loop;
```

SAVE COMMANDS

Save commands are used to store components in the working area into a component file or to store the active process into a process file. Save commands can be grouped into the following two commands:

I. SAVE COMPONENTS

FUNCTION: To copy the components data from the working area into a private component file.

GENERAL FORMAT: save components { all|(<component-list>)}
[override];

- GENERAL RULES:**
- 1) For every non existing specified component a warning message will be printed.
 - 2) For every specified component whose one or more parameters are unspecified a warning message will be printed and the component would not be saved.
 - 3) The private component file should be opened and the user should have the proper access.
 - 4) If the override option is not provided, for each specified component which already exists in the private component file a warning message will be printed.
 - 5) Only the component parameters will be copied into the private file, and the value types of parameters will not be copied.

EXAMPLE: save c(CO2,CO,New_comp);

II. SAVE PROCESS

FUNCTION: To store the current process network into the opened process file, under the specified name.

GENERAL FORMAT: save process <process> [override];

GENERAL RULES: 1) The process file should be opened and the user should have the proper access.

2) If the override option is not provided there must be no process with that name in the file.

3) If override option is provided, and there is another process in the file with the same name, the current process will override the old one.

EXAMPLE: save process test;

SPECIFY COMMANDS

Specify commands are used to specify data about a process element.

Specify commands can be grouped into the following six commands:

I. SPECIFY UNITS

FUNCTION: To supply data about units.

GENERAL FORMAT: specify units (<unit-list>) (<parameter -specification-list>)[<if-clause>];

GENERAL RULES: 1) The specified units must exist, and all must be the same type.

2) Specified parameters must be a valid parameter of that unit type.

3) Each expression should be evaluatable before the execution of the command.

EXAMPLE: specify u(HEATER) (U=20, A=40 'FT2');

II. SPECIFY STREAMS

FUNCTION: To specify phase (stream) parameters.

GENERAL FORMAT: specify streams (<stream-list>) [<phase-field>]
 (<parameter-specification-list>) [[<phase-field>]
 (<parameter-specification-list>)]* [<if-clause>];

GENERAL RULES: 1) The specified streams must exist, and all must be the same type.

2) If a phase-field is not specified the default would be phase 0 (total stream).

3) Each specified parameter must be a valid phase parameter of the specified phase of that stream type.

4) Each expression should be evaluatable before the execution of that command.

EXAMPLE: sp s (feed) (rate = 500 'scf/m') p1 (ratio = .2);

Parameter "rate" of phase zero and parameter "ratio" of phase one will be assigned.

III. SPECIFY COMPONENTS

FUNCTION: To supply data about components.

GENERAL FORMAT: specify components (<component-list>)
 (<parameter-specification-list>) [<if-clause>];

GENERAL RULES: 1) The specified components must exist, and all must be of the same type.

2) Each specified parameter must be a valid parameter of that component type.

- 3) Each expression should be evaluatable before the execution of the command.

EXAMPLE: sp c (k2so4) (density = 2,
%p4=2.4*c.k2so4.density); \$if c.k2so4.density is
unspecified the command is in error.\$

IV. SPECIFY FLOW

FUNCTION: To specify the flow parameters of the components flowing in a stream.

GENERAL FORMAT:

```
specify flow (<stream-list>)
[phase-field] (<flow-parameter-specification-list>)
[[phase-field] (<flow-parameter-specification-list>)]*
[<if-clause>];
```

- GENERAL RULES:
- 1) The specified streams must exist, and all must be of the same type.
 - 2) The specified components must be present in all streams.
 - 3) If phase-field is not specified the default would be phase 0 (total stream).
 - 4) Each parameter must be a valid flow parameter of the specified phase of that stream type.
 - 5) Each expression should be evaluatable before the execution of that command.

EXAMPLE: sp flow (FEED) (CO2(f=200));
Flow parameter "f" of component CO2 in phase 0 of stream feed is equal to 200.

V. SPECIFY FUNCTIONS

FUNCTION: To supply data about pre-defined functions.

GENERAL FORMAT: specify functions (<function-list>)
 (<parameter-specification-list>)
 [<if-clause>];

GENERAL RULES: 1) All the specified functions must:

- a) exist,
- b) be pre-defined, and
- c) be the same type.

2) Each specified parameter must be a valid parameter of that function type.

3) Each expression should be evaluatable before the execution of the command.

EXAMPLE: specify fn (t) (a0 = 1, a1 = 2);
 parameters a0 and a1 of pre-defined function t are assigned.

VI. SPECIFY VARIABLES

FUNCTION: To supply data about qualified variables or to create (if non-existing) simple variables and supply data for them.

GENERAL FORMAT: specify variables
 (<variable>=<expression>['dimension']
 [,<variable>=<expression>['dimension']]*
 [<if-clause>];

GENERAL RULES: 1) The variable could be simple or qualified.
 2) Valid dimensions are optional for qualified variables.

- 3) Each expression should be evaluatable before the execution of the command.
- 4) If a simple variable does not exist it will be created and its value will be assigned.

EXAMPLE:

a) sp v(u.a.t=10, x=u.a.t);

If u.a.t is unspecified, the command is in error, otherwise both u.a.t and x will be set to 10.

b) sp v(z=20, y=z);

If z is unknown, the command is in error, otherwise the value of z and y will be set to 20.

c) sp v (w=35) if (-1);

No action will be taken. If w is unknown, it will remain unknown.

STOP COMMAND

FUNCTION: To indicate the end of a repetitive execution of a group of commands headed by a repeat command.

GENERAL FORMAT: stop;

GENERAL RULES: 1) There must be an unterminated repetitive group of commands.

2) The execution of the corresponding group will terminate and that group will be deleted.

EXAMPLE: repeat for X = (10,20,30);

repeat for Y = (20,30,40);

.

.

stop;

The internal group of commands is only executed once (X=10, y = 20) and the group is deleted from the command sequence.

TERMINATE COMMANDS

These commands are used to terminate (delete) component private files and process files. Terminate commands are grouped to the two following commands.

I. TERMINATE COMPONENT

FUNCTION: To terminate one or more component private files.

GENERAL FORMAT: terminate component file (<file-list>);

- GENERAL RULES:
- 1) For every specified file the user will be prompted to enter the pathname of the file.
 - 2) Every file which the user has the required access will be terminated.
 - 3) Every file which the user has not the required access will produce an informatory system message.
 - 4) Every file which does not exist produces a warning message.
 - 5) If a specified file is opened it will be closed and then terminated.

II. TERMINATE PROCESS

FUNCTION: To terminate one or more process files.

GENERAL FORMAT: terminate process file (<file-list>);

- GENERAL RULES:
- 1) For every specified file the user will be prompted to enter the pathname of the file.

- 2) Every specified file which the user has the required access will be terminated.
- 3) Every specified file which the user has not the required access will produce an informatory system message.
- 4) Every specified file which does not exist produces a warning message.
- 5) If a specified file is opened it will be closed and then terminated.

EXAMPLE: terminate pr file (A,B);

UNSPECIFY COMMANDS

Unspecify commands are used to unspecify parameters and variables. The unspecify commands can be grouped into the following five commands:

I. UNSPECIFY UNITS

FUNCTION: To unspecify data about units.

GENERAL FORMAT: Type 1: unspecify units(<unit-list> {all|(<parameter-list>)});
Type 2: unspecify units all;

- GENERAL RULES:
- 1) Each unit in the list must exist.
 - 2) All units in the list must be the same type.
 - 3) All the specified parameters must belong to the unit type.
 - 4) For each unit in the list the parameters in the list will be unspecified.
 - 5) For Type 2, all parameters of all units will be unspecified.

EXAMPLE: unsp u all; All parameters of all units will be unspecified.

II. UNSPECIFY STREAMS

FUNCTION: To unspecify phase (stream) parameters.

GENERAL FORMAT: Type 1: unspecify streams (<stream-list>)
 [phase-field] {(<parameter-list>)|all}
 [[phase-field] {(<parameter-list>)|all}]*;
 Type 2: unspecify streams all;

- GENERAL RULES:
- 1) All the streams in the list must exist.
 - 2) All the streams in the list must be of the same type.
 - 3) If a phase-field is not specified phase 0 (total stream) will be assumed.
 - 4) All parameters in the list must belong to the specified phase of the stream type.
 - 5) For each stream in the list and for each specified phase, the parameters in the list will be unspecified.
 - 6) For Type 2, all phase parameters of all streams will be unspecified.

EXAMPLE: unspecify s(feed) phase 1 all;

All parameters of phase 1 of stream feed will be unspecified.

III. UNSPECIFY COMPONENTS

FUNCTION: To unspecify data about components.

GENERAL FORMAT: Type 1: unspecify components (<component-list>) {(<parameter-list>)|all};

Type 2: unspecify components all;

- GENERAL RULES:
- 1) All components in the list must exist.
 - 2) All components in the list must be of the same type.
 - 3) All the specified parameters must belong to the component type.
 - 4) For each component in the list the parameters in the list will be unspecified.
 - 5) For Type 2, all parameters of all components will be unspecified.

EXAMPLE: unsp c (ch4) (tc,pc);

IV. UNSPECIFY FLOW

FUNCTION: To unspecify flow parameters.

GENERAL FORMAT: Type 1: unspecify flow (<stream-list>) [phase-field] {(<flow-parameter-list>) | all } [[phase-field] {(<flow-parameter-list>) | all }]*;

Type 2: unspecify flow all;

- GENERAL RULES:
- 1) All streams in the list must exist and must be of the same type.
 - 2) If a phase-field is not specified phase 0 (total stream) will be assumed.
 - 3) All specified components must be present in each specified stream.
 - 4) all specified parameters must be valid flow parameters of the specified phase of the stream type.

- 5) For each stream in the list and for each phase in the list the given flow parameters of the specified components will be unspecified.
- 6) For Type 2, flow parameters of all components in all phases of all streams will be unspecified.

EXAMPLE: unsp flow (s) (co2 (rate),co all);

Flow parameter "rate" of component co2 and all flow parameters of component co in the phase 0 of streams will be unspecified.

V. UNSPECIFY VARIABLES

FUNCTION: To unspecify data about qualified variables and delete simple variables.

GENERAL FORMAT: unspecify variables {(<variable-list>)|all};

GENERAL RULES: 1) For the "all" option all existing simple variables will be deleted.

2) For the list option all variables must exist.

EXAMPLE: unsp variables (X,u.HEAT.A) ;

Simple variable X will be erased and parameter A of unit HEAT will be unspecified.

VI. UNSPECIFY FUNCTIONS

FUNCTION: To unspecify data about pre-defined functions.

GENERAL FORMAT: Type 1: unspecify functions (<function-list> {all|(<parameter-list>)}) ;

Type 2: unspecify functions all;

GENERAL RULES: 1) All specified functions must

- a) exist,
 - b) be pre-defined, and
 - c) be the same type.
- 2) Each specified parameter must be a valid parameter of that function type.
 - 3) For each function in the list the specified parameters in the list will be unspecified.
 - 4) For Type 2, all parameters or all pre-defined functions will be unspecified.

EXAMPLE: unsp fn (h) all;

USE COMMAND

FUNCTION: To allow the user to instruct the system to use the specified methods of physical properties estimations.

GENERAL FORMAT: Type 1: use print;
 Type 2: use [property-name=<integer-number>]*;
 Type 3: use default;

GENERAL RULES: 1) For Type 1; all options in effect will be printed.

2) For Type 2; the integer-number represents the method to be used for estimating a physical property.

3) For Type 2; the system will override the already specified or default options and the specified procedures are in effect until:

- a) overridden by another use command,

b) initialized to default options when creating a new process (clear and leave commands result in creation of a new process).

c) initialized to those of a new process when loading a process. Note that when saving a process the options in effect are also saved.

4) For Type 3; all options will be initialized to default options.

Default options are established by the TBS system administrator.

EXAMPLES: use pvap=2 fvap=4;
use print;
use default;

D.6 PEL MESSAGESNOTATIONS IN MESSAGES

Many of the messages produced in this section contain symbols indicating where the system will insert information when it prints the messages. These symbols and notations are as follows:

- t,t1,t2,t3 : Text up to 16 characters
- n : An integer number
- <object> : unit, component, function, stream, flow,
variable, or process
- <command-verb>: A command verb such as create, delete, etc.
- <file> : public or private
- <value-status>: unspecified, assumed, specified, or calculated
- <clause> : The name of the clause such as "if", "by",
"to", etc.
- <msg> : A message printed by Multics System.

D.6.1 WARNING MESSAGESD.6.1.1 INFORMATORY WARNING MESSAGES

WARNING i 1 <object> "t" does not exist

Example:

**COMMAND: delete u(A);

**COMMAND: delete u(A,B);

WARNING i 1 Unit "A" does not exist.

WARNING i 2 <object> "t" already exists.

Example:

**COMMAND: crt c(C1);

**COMMAND: load c(C1);

WARNING i 2 component "C1" already exists.

WARNING i 3 component "t" cannot be erased. It is present in one or more of existing streams.

Example

**COMMAND: crt flow (S1)(CO2,CO);

**COMMAND: delete component (CO2,CO);

WARNING i 3 component "CO2" cannot be erased. It is present in one or more of existing streams.

WARNING i 3 component "CO" cannot be erased. It is present in one or more of existing streams.

WARNING i 4 function "t" cannot be erased. It is referred in an existing user-defined function.

Example

**COMMAND: crt fn (H(X1, X2)=X1/X2);

**COMMAND: crt fn (G(Y1,Y2)=H(Y1/Y2,Y2/Y1)+4);

**COMMAND: delete fn (H,G);

WARNING i 4 function "H" cannot be erased. It is referred in an existing user-defined function.

**COMMAND: delete fn (H);

**COMMAND:

WARNING i 5 variable "t" have been already unspecified in this command.

Example

**COMMAND: unsp v(u.A.T,X,Y,X);

WARNING i 5 variable "X" have been already unspecified in this command.

WARNING i 6 maximum number of components (n) have been created. No more components including "t" can be created.

Example

**COMMAND: crt c (C3,C5,C12);

WARNING i 6 maximum number of components (20) have been created. No more components including "C5" can be created.

Explanation

Maximum number of components has been set to 20 by the user at the time of initiation of this process. 19 components have already been created ,so this command only has created component C3.

WARNING i 7 component "t" is not present in stream "t"

Example

**COMMAND: delete f(S1)(CO2);

WARNING i 7 component "CO2" is not present in stream
"S1".

WARNING i 8 component "t" is already present in stream "t1".

Example

**COMMAND: crt f(S1)(CO);

WARNING i 8 component "CO" is already present in
stream "S1".

WARNING i 9 stream "t" does not accept component type "t1".

Example

**COMMAND: crt c(C10) type=XYZ;

**COMMAND: crt f(S1)(C10);

WARNING i 9 stream "S1" does not accept component
type "XYZ".

WARNING i 10 unit is already disconnected at "t"

Example

**COMMAND: disconnect u A at all;

**COMMAND: disconnect u A at (IN);

WARNING i 10 unit is already disconnected at "IN".

WARNING i 11 unit is already connected at "t" to stream "t1".

Example

**COMMAND: cnct u A at IN=S1 IN=S2;

WARNING i 11 unit is already connected at "IN" to stream "S1".

WARNING i 12 stream "t" is of type "t1". A stream of type "t1" is required for connection "t3".

Example

**COMMAND: crt s(S9) type=XYZ;

**COMMAND: cnct u A at all=,S9;

WARNING i 12 stream "S9" is of type "XYZ". A stream of type "std" is required for connection "IN2"

Explanation

The above command requests that stream S9 to be connected to the 2nd connection of unit A which is "IN2". That connection can only accept a stream of type "std".

WARNING i 13 stream "t" is already connected at destination to "t1" of unit "t2".

Example

**COMMAND: cnct u B at IN=S1;

**COMMAND: cnct u C at IN2=S1;

WARNING i 13 stream "S1" is already connected at destination to "IN" of unit "B".

WARNING i 14 stream "t" is already connected at source to "t1" of unit "t2".

Example

**COMMAND: cnc t u B at OUT1=S3, OUT2=S3;

WARNING i 14 stream "S3" is already connected at source to "OUT1" of unit "B".

WARNING i 15 this external repeat command is not executable.

Example

**COMMAND: r for X from (10) to (1);

WARNING i 15 this external repeat command is not executable.

WARNING i 16 component "t" type "t1" is present in private file "t2" and override option is not specified.

Example

**COMMAND: load c (C1,C2,C3) type=XYZ public;

**COMMAND: save c (C1,C4);

WARNING i 16 component "C1" type "XYZ" is present in private file "xxx" and override option is not specified.

WARNING i 17 component "t" type "t1" is not found in <file> file "t2".

Example:

**COMMAND: load c (c10,c2);

WARNING i 17 component "c10" type "xyz" is not found in private file "xxx".

WARNING i 18 one or more parameters of component "t" are unspecified. Component cannot be saved.

Example

**COMMAND: unspecify c (c1,c2)(tc,pc);

**COMMAND: save c (c3,c1);

WARNING i 18 one or more parameters of component "c1" are unspecified. Component cannot be saved.

WARNING i 19 process file "t" does not exist.

Example:

**COMMAND: terminate pr (x);

(User is prompted for the pathname of the process file "x".)

WARNING i 19 process file "x" does not exist.

WARNING i 20 no room is left for adding any component including "t" to the private file "t1".

Example:

****COMMAND:** save c (c1,c2,c3,c10);

*****WARNING***** i 20 no room is left for adding any component including "c3" to the private file "xxx".

*****WARNING***** i 22 maximum number of component types (n) has been added to the private file "t". Therefore no more components of the new type "t1" can be added to the file.

Examples:

****COMMAND:** save c (c10, c12);

*****WARNING***** i 22 maximum number of component types (20) has been added to the private file "xxx". Therefore no more components of the new type "yyy" can be added to the file.

*Component c10 is not saved.

****COMMAND:** inc c (c11,c12,c13) type = yyy;

*****WARNING***** i 22 maximum number of component types (20) has been added to the private file "xxx". Therefore no more components of the new type "yyy" can be added to the file.

Explanation:

At the time of creating private file "xxx" maximum number of component types has been specified to be 20. Therefore only the maximum of 20 component types can be added to the file.

WARNING i 23 process "t" already exists and override option is not specified.

Example:

**COMMAND: save pr test;

WARNING i 23 process "test" already exists and override option is not specified.

WARNING i 24 process "t" is not found.

Example:

**COMMAND: delete (test, test1);

WARNING i 24 process "test1" is not found.

WARNING i 25 comment terminator, "\$", is missing. One is assumed.

Example:

**COMMAND: crt u \$ this is a comment.

WARNING i 25 comment terminator, "\$", is missing.
One is assumed.

**Continue: (A,B);

WARNING i 26 stream "t" is of type "t1" which is not allowed to contain any component.

Example:

**COMMAND: crt s (feed) type = air;

**COMMAND: crt f (feed) (CO,CO₂);

WARNING i 26 stream "feed" is of type "air" which is not allowed to contain any component.

WARNING i 27 text following the ";" is ignored.

Example:

**COMMAND: crt u (A,B); this is extra.

WARNING i 27 text following the ";" is ignored.

WARNING i 28 argument "t" has not been referred to in expression defining function "t1".

Example:

**COMMAND: crt fn (xyz(x1,x2,x3) = x1 + x2);

WARNING i 28 argument "x3" has not been referred to in expression defining function "xyz".

D.6.1.2 SEVERE WARNING MESSAGES

The general format of severe warning messages is as follows:

```
***WARNING*** s n in evaluating{the <clause> clause of | an expression in}
the <command-verb> [<object>] command, the following has
been detected:
```

```
---Message n---
```

The result of the expression has been assumed to be 0.0

where: s indicates that the message is of severe type. n is the severe warning message number.

Message n is one of the following:

1. The absolute value of the argument of acos or asin built-in function is greater than 1.
2. The absolute value of the argument of atanh built-in function is not less than 1.
3. The argument of log, log2, or log10 built-in function is not greater than 0.
4. The second argument of mod built-in function is 0.
5. The argument of sqrt built-in function is negative.
6. Both arguments of atan or atand built-in function are 0.
7. A division by 0.
8. An overflow.
9. An underflow.
10. Zero has been raised to a non-positive number.
11. One or more parameters of pre-defined function "t" are unspecified.

Example:

```
**COMMAND: repeat for y from (1) to (-1) by (-1).
```

```
**COMMAND: sp v (x = sqrt(y)**2);
```

```
**COMMAND: p v(y,x);
```

```
    y = 1
```

```
    x = 1
```

```
**COMMAND: loop;
```

```
    y = 0
```

```
    x = 0
```

```
***WARNING*** s 5 in evaluating an expression in the  
specify variable command the following has been detected:
```

```
The argument of sqrt built-in function is negative.
```

```
The result of the expression has been assumed to be 0.0.
```

```
    y = -1
```

```
    x = 0
```

D.6.2 ERROR MESSAGESD.6.2.1 INFORMATORY ERROR MESSAGES

*****ERROR***** i 1 bad input or line greater than n characters.

Explanation:

When inputting a line, either a transmission error has been detected or line size is greater than n characters. The value of the n depends on the nature of the input line. A command may consist of an unlimited number of lines but each line should be less than 262 characters. Component file's remark should be less than 100 characters.

Example:

****Continue:** (user has entered a line greater than 262 characters)

*****ERROR***** i 1 bad input or line greater than 262 characters.

****Reenter the line:**

*****ERROR***** i 2 illegal qualified variable. First element is not valid.

Example:

****COMMAND:** sp v(d.t.f = 2);

sp v(d.t.f = 2);
\$

*****ERROR***** i 2 illegal qualified variable. First element is not valid.

****Reenter the input:** sp v(u.t.f = 2);

Explanation:

d is invalid as a first element of a qualified variable.
 First element of a qualified variable should be one of the
 following:

u, unit, units

c, component, components

fn, function, functions

s, stream, streams

f, flow

ERROR i 3 the closing " is missing.

Example:

```
**COMMAND: crt u ("HEAT EXCHANGER");
           $
```

ERROR i 3 the closing " is missing.

**Reenter the line: crt u ("HEAT EXCHANGER");

ERROR i 4 exponent is greater than 127 or less than -128.

Example:

```
**COMMAND: sp v(x = 12.4e+129);
           sp v(x = 12.4e+129);
           $
```

ERROR i 4 exponent is greater than 127 or less than
 -128

**Reenter the line:

ERROR i 5 illegal qualified variable. Excess point or missing element, or one of the elements is an invalid identifier.

Example:

```
**COMMAND:  sp v(x = .u.t.f);
             sp v(x = .u.t.f);
```

\$

ERROR i 5 illegal qualified variable. Excess point or missing element, or one of the elements is an invalid identifier.

ERROR i 6 identifier, number, or dimension is more than 16 characters.

Examples:

```
1.  crt u ("A VERY LONG IDENTIFIER");
     $
```

```
2.  100 'A VERY LONG DIMENSION';
     $
```

```
3.  sp v(x = 123456789.12345e+2);
     $
```

Explanation:

1. Identifier more than 16 characters.
2. Dimension more than 16 characters.
3. Number more than 16 characters.

ERROR i 7 empty dimension field or identifier field.

Example:

```
1.  crt u ("",A);
     $
```

```
2.  sp v(U.A.T = 10'');
     $
```


Explanation:

1. An identifier which is enclosed in a pair of double quotation marks or a dimension field should satisfy the following restrictions:

1. Should not be null.
2. Should not contain any tab settings.
3. All blanks following the last non-blank character should be ignored.
4. Except for the trimmed blanks, they can only contain 1 to 16 characters.

ERROR i 8 illegal unit qualified variable. Number of elements should be 3.

Example:

```
**COMMAND: sp v(u.UNAME.UPARM.XTRA = 4);
                $
```

Explanation:

The above qualified variable has 4 elements. A unit qualified variable should have 3 elements. First element should be u, unit, or units. The second element is the unit identifier. The third element is the referred parameter.

ERROR i 9 illegal component qualified variable. Number of elements should be 3.

Example:

```
**COMMAND: sp v(component.CO2.TC.XTRA);
                $
```

Explanation:

The above qualified variable has 4 elements. A component qualified variable should have 3 elements. The first element should be c, component, or components. The second element is the component identifier. The third element is the referred parameter.

ERROR

i 10 illegal function qualified variable. Number of elements should be 3.

Example:

```
**COMMAND: sp v(FN.A = 4);
                $
```

Explanation:

The above qualified variable has 2 elements. A function qualified variable should have 3 elements. The first element should be fn, function, or functions. The second element is the function identifier. The third element is the referred parameter (coefficient).

ERROR

i 11 illegal stream qualified variable. Number of elements should be 4.

Example:

```
**COMMAND: sp v(x = s.SNAME.s.PHASEPARM.XTRA)
                $
```

Explanation:

The above qualified variable has 5 elements. A stream qualified variable should have 4 elements. The first element should be s, stream, or streams. The second element is the stream identifier. The third element represents the phase field. The fourth element is the referred parameter.

ERROR i 12 illegal flow qualified variable. Number of elements should be 5.

Example:

```
**COMMAND:  sp v(f.SNAME.0.CO2 = 3);
                $
```

Explanation:

The above qualified variable has 4 elements. A flow qualified variable should have 5 elements. The first element should be f or flow. The second element is the stream identifier. The third element represents the phase field. The fourth element is the desired component. The fifth element is the referred flow parameter.

ERROR i 13 illegal appearance of not sign (^). It is only allowed in combination with equal sign (^=).

Example:

```
**COMMAND:  sp v(x = 1) if (^2)
                $
```

*****ERROR***** i 14 an identifier or dimension is not allowed to contain any tab settings.

Example:

```
**COMMAND: crt u ("          A");
                $
```

```
**COMMAND: sp v(U.A.T. = 2 '          DIM');
                $
```

Explanation:

An identifier enclosed in a pair of " or a dimension enclosed in a pair of ' cannot contain any tab settings.

*****ERROR***** i 15 the closing ' is missing.

Example:

```
**COMMAND: sp v(U.A.T = 'DIM);
                $
```

*****ERROR***** i 16 "t" is an invalid integer number. A valid integer number consists of 1 to 5 digits.

Example:

```
**Enter maximum number of components: 20.
```

```
***ERROR*** i 16 "20." is an invalid integer number. A
valid integer number consists of 1 to 5 digits only.
```

```
**Reenter the number: 20
```

Explanation:

This error occurs when inputting an integer number and the number does not consist of 1 to 5 digits.

ERROR i 17 the number should be not greater than n.

Example:

**Enter maximum number of components (0 to 200): 250

ERROR i 17 the number should be not greater than 200.

**Reenter the number:

Explanation:

This error occurs when inputting an integer number and the number is greater than a preset maximum limit.

ERROR i 18 "t" is an invalid number.

Example:

**COMMAND: read v(x);

x:12.A4

ERROR i 18 "12.A4" is an invalid number.

**Reenter the input:

ERROR i 19 your response should be yes or no.

Example:

**If you wish to continue enter yes, otherwise no: what

ERROR i 19 your response should be yes or no.

**Reenter please:

ERROR i 20 dimension 't' is not allowed for variables or dimensionless parameters.

Example:

**COMMAND: read v(x,u.A.T);

**Enter the following variables:

x: 10 'DEGK'

ERROR i 20 dimension 'DEGK' is not allowed for
variables or dimensionless parameters.

**Reenter the input:

ERROR i 21 't' is an invalid dimension for the specified
parameter.

Example:

**COMMAND: read v(u.A.T);

**Enter the following variables:

u.A.T = 10 'DEG-DUMMY'

ERROR i 21 'DEG-DUMMY' is an invalid dimension for
the specified parameter.

**Reenter the input:

ERROR* i 22 text beginning "t" is excess.

Example:

**COMMAND: reada v(x);

**Enter the following variables:

x: 10 20 A

ERROR i 22 text beginning "20" is excess.

**Reenter the input:

D.6.2.2 SEVERE ERROR MESSAGES

ERROR s 1 "t". After "t1" a ";" is expected.

Example:

**COMMAND: crt u(A,B) HEAT_EX;

ERROR s i "HEAT EX". After ")" a ";" is expected.

*Command ingored.

**COMMAND: crt u(A,B) type = HEAT_EX

**Continue: \$ or if HEAT_EX is the default type \$

**Continue: \$ it is equivalent to crt u(A,B); \$

**Continue: ;

**COMMAND:

ERROR s 3 "t". After "t1" a ",," is expected.

Example:

**COMMAND: crt u(A B);

ERROR s 3 "B". After "A" a ",," is expected.

*Command ignored.

**COMMAND: crt u(A,B);

ERROR s 5 "t". After "t1" a ")" is expected.

Example:

**COMMAND: sp v(x = 10) if (y > 12, x < 3;

ERROR s 5 "1". After "12" a ")" is expected.

*Command ignored.

**COMMAND: sp v(x = 10) if (y > 12 & x < 3;

ERROR s 5 ";". After "3", a ")" is expected.

*Command ignored.

**COMMAND: sp v(x = 12) if (y > 12 & x < 3);

**COMMAND:

ERROR s 6 "t" is an invalid option for t. It must be a non-zero integer number not greater than n.

Example:

**COMMAND: use FLIQ = 6;

ERROR s "6" is an invalid option for FLIQ. It must be a non-zero integer number not greater than 4.

ERROR s 7 "t" is an invalid argument. It must be an integer number not greater than 99999.

Example:

**COMMAND: calculate unit (A(+3));

ERROR: s 7 "+" is an invalid argument. It must be an integer number not greater than 99999.

*Command ignored.

**COMMAND: calculate u(A(3));

ERROR s 8 "t" is an invalid dimension type. It must be an integer number not greater than n.

Example:

****COMMAND:** print dimension A;

*****ERROR***** s 8 "A" is an invalid dimension type. It must be an integer number not greater than 20

*****ERROR***** s 9 "t" is an invalid property type. It must be a non-zero integer number not greater than n.

Example:

****COMMAND:** print property B;

*****ERROR***** s 9 "B" is an invalid property type. It must be a non-zero integer number not greater than 6.

*****ERROR***** s 10 "t". After "t1" a "=" is expected.

Example:

****COMMAND:** sp v(x 12, y 20);

*****ERROR***** s 10 "12". After "x" a "=" is expected.

*Command ignored.

****COMMAND:** sp v(x = 12, y 20);

*****ERROR***** s 10 "20". After "y" a "=" is expected.

*Command ignored.

****COMMAND:** sp v(x = 12, y = 20);

****COMMAND:**

*****ERROR***** s 16 "t". After "t1" a "(" is expected.

Example:

****COMMAND:** crt u A;

*****ERROR***** s 15 "A". After "u" A "(" is expected.

*Command ignored.

**COMMAND: crt U(A);

**COMMAND:

ERROR s 19 "t". After "t1" "at" is expected.

Example:

**COMMAND: cnet unit A IN = S1;

ERROR s 19 "IN". After "A" "at" is expected.

*Command ignored.

**COMMAND: cnet unit A at IN = S1;

**COMMAND:

ERROR s 20 "t". After "t1" "for" is expected.

**COMMAND: repeat x from (1) to (10);

ERROR s 20 "x". After "repeat" "for" is expected.

**COMMAND: repeat for x from (1) to (10);

ERROR s 21 "t". After "t1" "to" is expected.

Example:

**COMMAND: r for x from (1) by (2) to (10);

ERROR s 21 "by". After ")" "to" is expected.

*Command ignored.

**COMMAND: r for x from (1) to (10) by (2);

**COMMAND:

ERROR s 23 "t". After "t1" "file" is expected.

Example:

**COMMAND: open c (x);

ERROR s 23 "(" . After "c" "file" is expected.

*Command ignored.

**COMMAND: open c file (x);

ERROR s 25 "t". After "t1" an identifier is expected.

Example:

**COMMAND: crt u(A,B,);

ERROR s 25 ")" . After "," An identifier is expected.

*Command ignored.

**COMMAND: crt u(A,B,C);

ERROR s 26 "t" is an unknown level number for calculation of
<object> t1.

Example:

**COMMAND: calc u(A(10,1));

ERROR s 26 "10" is an unknown level number for
calculation of unit A.

*Command ingnored.

**COMMAND:

ERROR s 27 "t" is an unknown unit.

Example:

**COMMAND: sp v(x = u.E.T);

ERROR s 27 "E" is an unknown unit.

*Command ignored.

**COMMAND:

ERROR s 28 "t" is an unknown component.

Example:

**COMMAND: sp c(c1,c2) (tc = 2, pc = 4);

ERROR s 28 "c1" is an unknown component.

*Command ignored.

**COMMAND:

ERROR s 29 "t" is an unknown pre-defined function.

Example:

**COMMAND: print fn(A,B) all;

ERROR s 29 "A" is an unknown pre-defined function.

*Command ignored.

**COMMAND:

ERROR s 30 "t" is an unknown stream.

Example:

**COMMAND: sp s(s1,s2)(t = 100);

ERROR s 30 "s1" is an unknown stream.

*Command ignored.

ERROR s 32 "t" is an unknown variable.

Example:

**COMMAND: print v(xx,yy);

ERROR s 32 "yy" is an unknown variable.

*Command ignored.

ERROR s 34 "t" is an invalid <object> type.

Example:

**COMMAND: crt u(A) type = ADUMMY;

ERROR s 34 "ADUMMY" is an invalid unit type.

*Command ingored.

**COMMAND: crt u(A) type = HEAT_EX;

ERROR s 35 "t" is a pre-defined function which may not appear in an arithmetic expression.

Example:

**COMMAND: sp(x = H(2,3) + 4);

ERROR s 35 "H" is a pre-defined function which may not appear in an arithmetic expression.

Explanation:

There is no evaluating routine associated with this type of pre-defined function.

ERROR s 38 "t" is an invalid unit parameter.

Example:

**COMMAND: sp u(U1)(TT = 200);

ERROR s 38 "TT" is an invalid unit parameter.

ERROR s 39 "t" is an invalid component parameter.

Example:

**COMMAND: sp v(c.CO2.TK = 10);

ERROR s 39 "TK" is an invalid component parameter.

ERROR s 40 "t" is an invalid function parameter.

Example:

**COMMAND: sp fn(K,K2)(AKI = 10);

ERROR s 40 "AKI" is an invalid function parameter.

ERROR s 41 "t" is an invalid stream parameter.

Example:

**COMMAND: sp s(s1,s2)(TT = 25);

ERROR s 41 "TT" is an invalid stream parameter.

ERROR s 42 "t" is an invalid flow parameter.

Example:

**COMMAND: unspecify flow (S1,S2)(CO2(FF,X));

ERROR s 42 "FF" is an invalid flow parameter.

ERROR s 43 "t" is an invalid connection of unit type t1.

Example:

**COMMAND: connect u A at INN=S1;

ERROR s 43 "INN" is an invalid connection of unit type HEAT_EX.

ERROR s 44 "t" is an invalid property name.

Example:

**COMMAND: use FFLIQ = 2;

ERROR s 44 "FFLIQ" is an invalid property name.

ERROR s 45 "t" is an invalid profile parameter.

Example:

**COMMAND: profile S = 14;

ERROR s 45 "S" is an invalid profile parameter.

ERROR s 46 "t" is an invalid print option for <object>.

Example:

**COMMAND: print unit (A,B)PARM;

ERROR s 46 "PARM" is an invalid print option for unit.

ERROR s 47 "t" is an invalid phase field. Valid phase fields are: pn, phasen, s, stream, and streams, where n is an integer number less than or equal to the number of phases

of the specified stream type.

Example:

**COMMAND: sp v(x = s.SNAME.PP.T);

ERROR s 47 "PP" is an invalid phase field. Valid phase fields are pn, pahsen, s, stream, and streams, where n is an integer number less than or equal to the number of phases of the specified stream type.

ERROR s 49 "t" already has been created.

Example:

**COMMAND: crt fn (ADD (x,y) = x + y);

ERROR s 49 "ADD" already has been created.

*Command ignored.

ERROR s 50 "t" has appeared more than once in the list.

Example:

**COMMAND: crt fn(SUB(x,y) = x - y, SUB(x,xx) = x-xx);

ERROR s 50 "SUB" has appeared more than once in the list.

*Command ignored.

ERROR s 51 "t" is not the same type as "t1".

Example:

**COMMAND: sp u(U1,U2)(t = 100);

ERROR s 51 "U2" is not the same type as "U1".

ERROR s 52 "t" is an unknown function.

Example:

**COMMAND: sp v(x = xyz(10,11));

ERROR s 52 "xyz" is an unknown function.

ERROR s 55 "t" is an invalid command verb.

Example:

**COMMAND: command;

ERROR s 55 "command" is an invalid command verb.

ERROR s 56 "t" is an invalid command object for t1 command.

Example:

**COMMAND: describe crt ff;

ERROR s 56 "ff" is an invalid command object for crt command.

ERROR s 57 dimension 't' is an invalid dimension for the specified parameter.

Example:

**COMMAND: sp v(u.a.t = 10 'deg-dummy');

ERROR s 57 dimension 'deg-dummy' is an invalid dimension for the specified parameter.

ERROR s 58 dimension 't' is not allowed for variables, or dimensionless parameters.

Example:

****COMMAND:** sp v(x = 10 'degf');

*****ERROR***** s 58 dimension 'degf' is not allowed for variables, or dimensionless parameters.

*****ERROR***** s 59 component "t" is not present in stream "t1".

Example:

****COMMAND:** sp v(x = f.s1.po.co2.x);

*****ERROR***** s 59 component "co2" is not present in stream "s1".

*****ERROR***** s 60 transfer point "t" does not appear earlier in the list.

Example:

****COMMAND:** calculate u(a,b,v(1,d));

*****ERROR***** s 60 transfer point "d" does not appear earlier in the list.

*****ERROR***** s 61 routine for calculating <object> t is not implemented yet.

Example:

****COMMAND:** calc u(a,b);

*****ERROR***** s 61 routine for calculating unit b is not implemented yet.

ERROR s 62 this command is not allowed to appear inside the repeat-loop group.

Example:

**COMMAND: r for x from (1) to (u.a.t);

**COMMAND: delete u (a);

ERROR s 62 this command is not allowed to appear inside the repeat-loop group.

*Command ignored.

ERROR s 63 no active repeat command precedes this command.

Example:

**COMMAND: loop;

ERROR s 63 no active repeat command precedes this command.

ERROR s 64 incorrect number of arguments for <object> t. It requires at least n arguments.

Example:

**COMMAND: calc u(b,c(1,2,b));

ERROR s 64 incorrect number of arguments for unit c. It requires at least 3 arguments.

ERROR s 65 incorrect number of arguments for <object> t. It is allowed to have n arguments or less.

Example:

****COMMAND:** calc unit (a(1,2,3,4));

*****ERROR***** s 65 incorrect number of arguments for unit
a, It is allowed to have 2 arguments or less.

*****ERROR***** s 66 sdigit must be a non-zero integer number not greater
than 14.

Example:

****COMMAND:** prof sdigit = 16;

*****ERROR***** s 66 sdigit must be a non-zero integer number
not greater than 14.

*****ERROR***** s 67 ddigit must be an integer number not greater than
sdigit.

Example:

****COMMAND:** prof ddigit = 14 sdigit = 12;

*****ERROR***** s 67 ddigit must be an integer number not
greater than sdigit.

*****ERROR***** s 68 dflag must be 0, 1, 2, or 3.

Example:

****COMMAND:** prof dflag = +1;

*****ERROR***** s 68 dflag must be 0, 1, 2, or 3.

*****ERROR***** s 69 output must be 0 or 1.

Example:

****COMMAND:** profile output = 2;
*****ERROR***** s 69 output must be 0 or 1.

*****ERROR***** s 70 input must be 0 or 1.

Example:

****COMMAND:** profile input = 2;
*****ERROR***** s 69 input must be 0 or 1.

*****ERROR***** s 71 invalid expression.

Example:

****COMMAND:** sp v(x = a b);
*****ERROR***** s 71 invalid expression.

*****ERROR***** s 72 invalid expression. One or more ")" is missing.

Example:

****COMMAND:** sp v(x = (a*b)/(a-b, z =2);
*****ERROR***** s 72 invalid expression. One or more ")" is missing.

*****ERROR***** s 73 invalid expression. Unspecified qualified variable.

Example:

****COMMAND:** sp v(x = u.a.t).
*****ERROR***** s 73 invalid expression. Unspecified qualified variable.

ERROR s 74 invalid expression. Wrong number of arguments for function "t".

Example:

**COMMAND: crt fn(xyz(x1,x2) = x1 + x2);

**COMMAND: sp v(x = xyz(1,2,3) + 4.2);

ERROR s 74 invalid expression. Wrong number of arguments for function "xyz".

ERROR s 75 no <object> type exists.

Example:

**COMMAND: crt fn(wxyz);

ERROR s 75 no function type exists.

Explanation:

The above command requests that a pre-defined function of type wxyz be created. And since there is no pre-defined function type the above error message has resulted.

ERROR s 76 number of specified parameters in "all" option exceeds the number of parameters, or excess ",".

Example:

**COMMAND: sp u(A) (all = 1,2,3,4,5,6);

ERROR s 76 number of specified parameters in "all" option exceeds the number of parameters or excess ",".

Explanation:

Unit A has only 4 parameters while 6 parameters have been specified in all option.

ERROR s 77 number of specified streams in all option exceeds the number of connections, or excess ",".

Example:

**COMMAND: unit u a at all = s1,,,s4,s5,s6,,,s8,,,s10;

ERROR s 77 number of specified streams in all option exceeds the number of connections or excess ",".

ERROR s 90 no component <file> file is opened.

Example:

**COMMAND: close c private file;

**COMMAND: include c (c1,c2,c3);

ERROR s 90 no component private file is opened.

ERROR s 91 no process file is opened.

Example:

**COMMAND: close process file;

**COMMAND: load pr test;

ERROR s 91 no process file is opened.

ERROR s 92 component type "t" is not found in <file> file t1

Example:

**COMMAND: open c private file (xxx);

(User is prompted for the pathname of the file.)

**COMMAND: load c (c1,c2,c3) type = xyz private;

ERROR s 92 component type "xyz" is not found in private file xxx.

D.6.2.3 CALCULATING ERROR MESSAGES

*****ERROR***** c 1 for level n calculation of unit "t" it should { be | not be } connected at "t1".

Example:

****COMMAND:** calc u(a,b);

*Entering routine aaa for level 1 calculation of unit "a"

*****ERROR***** c 1 for level 1 calculation of unit "b", it should be connected at "in3".

*Due to above error(s) the execution of this command has been terminated.

****COMMAND:**

*****ERROR***** c 2 for level n calculation of <object> "t" parameter "t1" of <object> "t" should not be <value-status>

Example:

****COMMAND:** calc unit (a(2));

*****ERROR***** c 2 for level 2 calculation of unit "a" parameter "x" of unit "a" should not be unspecified.

*****ERROR***** c 3 for level n calculation of <object> "t" parameter "t1" of phase n1 of stream "t2" should not be <value-status>

Example:

****COMMAND:** calc u(A);

*****ERROR***** c 3 for level 1 calculation of unit "a" parameter "t" of phase 0 of stream "x" should not be

specified.

Explanation:

Stream "x" is connected to unit A and parameter "t" of the stream is output of the unit calculation.

ERROR c 4 for level n calculation of < object > "t" flow parameter "t1" of component "t2" in phase n1 of stream "t3" should not be < value-status >

Example:

**COMMAND: calc u(A);

ERROR c 4 for level 1 calculation of unit "a" flow parameter "f" of component "nc4" in phase 0 of stream "x" should not be unspecified.

ERROR c 4 for level 1 calculation of unit "a" flow parameter "f" of component "nc5" in phase 0 of stream "x" should not be unspecified.

*Due to above error(s) the execution of this command has been terminated.

**COMMAND:

ERROR c 5 for level n calculation of < object > "t" parameter "t1" of component "t2" should not be < value-status >

Example:

**COMMAND: calc unit (b);

ERROR c 5 for level 1 calculation of unit "b"

parameter "tc" of component "nc4" should not be unspecified.

Explanation:

Component "nc4" is flowing in a stream connected to unit "b".

ERROR

c 6 for level n calculation of <object> "t" one or more errors are detected by the calculating routine t1.

**COMMAND: calc unit (b,a);

*Entering routine bbb for level 1 calculation of unit "b"

*Entering routine aaa for level 1 calculation of unit "a"

AAA: method of convergence has failed.

ERROR c 6 for level 1 calculation of unit "a" one or more errors are detected by the calculating routine aaa.

ERROR

c 7 for level n calculation of <object> "t" an error has been detected during the execution of the calculating routine which has not been handled by that routine.

Example:

**COMMAND: calc unit (c);

*Entering routine ccc for level 1 calculation of unit "c"

ERROR c 7 for level 1 calculation of unit "c" an error has been detected during the execution of the calculating routine which has not been handled by that routine.

Explanation:

The error may be due to underflow, overflow, zero divide or a similar condition in the calculating routine.

ERROR c 8 for level n calculation of <object> "t" user has interrupted the execution.

Example:

**COMMAND: calc unit (a,b);

*Entering routine aaa for level 1 calculation of unit "a"

(user has hit the interrupt button)

(MULTICS ready message)

(user enters "pi" to return to system)

ERROR c 8 for level 1 calculation of unit "a" user has interrupted the execution.

*Due to above error(s) the execution of this command has been terminated.

D.6.3 SYSTEM MESSAGES

D.6.3.1 INFORMATORY SYSTEM MESSAGES

SYSTEM i 1 component private file "t" does not exist. cdbsys: msg

Example:

**COMMAND: terminate c file (xx).

(User is prompted for the pathname of the file.)

SYSTEM i 1 component private file "xx" does not exist.

cdbsys: entry not found.

SYSTEM i 2 component file "t" is not compatible with current TBS. Number of parameters for component type "t1" is n1 in this file while it is n2 in the current TBS.

Example:

**COMMAND: save components (c2);

SYSTEM i 2 component private file "xyz" is not compatible with current TBS. Number of parameters for component type "std" is 5 in this file while it is 6 in the current TBS.

*Component "c2" is not saved.

**COMMAND:

SYSTEM i 3 you have deleted segment "t" outside of this program.
pobsys: msg.

Example:

**COMMAND: delete pr test;

SYSTEM i 3 you have deleted segment "xtest1.GPES"
outside of this program. pbsys: entry not found.

**COMMAND: load pr test10;

SYSTEM i 3 you have deleted segment xtest101.GPES
outisde of this program. pbsys: entry not found.

*Command ignored.

**COMMAND:

SYSTEM i 4 no text is given for "t"

Example:

**COMMAND: describe unit;

SYSTEM i 4 no text is given for "unit".

SYSTEM i 5 invalid argument "t". Valid arguments are: brief or
bf.

Example:

When entering the system: pel b

SYSTEM i 5 invalid argument "b".

Valid arguments are: brief or bf.

SYSTEM i 6 invalid pathname or no access for terminating
component file "t". cdbsys: msg.

Example:

**COMMAND: terminate component file (xx);

(User is prompted for the pathname of the file);

SYSTEM i 6 invalid pathname or no access for
terminating component file "xx". cdbsys: bad syntax in
the pathname

SYSTEM i 7 invalid pathname or no access for terminating process
file "t". pdbsys: msg.

Example:

Similar to that given for the above message.

D.6.3.2 SEVERE SYSTEM MESSAGES

SYSTEM s 1 unit type "t" is not present in current TBS.

Example:

**COMMAND: load pr test4;

*Process "test4" has been created at: 4/27/78 1141.7 est thu

by system: GPES Serial_No: 1 Compat_level: 1

and TBS: test Serial_No: 1 Compat_level: 1

and has not been accessed by any other incompatible system since.

SYSTEM s 1 unit type "xyz" is not present in current TBS.

*Command ignored.

Explanation:

The current generation of the TBS does not contain the template of unit type xyz and hence is incompatible with the one under which process test4 was created. The TBS Administrator has failed to update the Compat_level of the new version of the TBS so that the system can recognize this incompatibility earlier.

SYSTEM s 2 component type "t" is not present in current TBS.

Example:

**COMMAND: load pr test4;

*Process "test4" has been created at: 4/27/78 1141.7 est thu

by system: GPES Serial_No: 1 Compat_level: 1
 and TBS: test Serial_No: 1 Compat_level: 1
 and it has been accessed by other incompatible systems
 since.

Therefore it may not be compatible with current system:

GPES Serial_No: 1 Compat_level: 1

and TBS: test Serial No: 3 Compat_level: 2.

**If you wish to continue loading enter yes, otherwise no:

yes

SYSTEM s 2 component type "xzy" is not present in
 current TBS

*Command ignored.

**COMMAND:

SYSTEM s 3 function type "t" is not present in current TBS.

Example:

Similar to one given for s1 or s2.

SYSTEM s 4 stream type "t" is not present in current TBS.

Example:

Similar to one given for s1 or s2.

SYSTEM s 5 segment "stext" is not found or you are not authorized
 to use the system. pel: msg.

Example:

This message may be produced at the beginning of the session: pel brief

SYSTEM s 5 segment "stext" is not found or you are not authorized to use the system. pel: entry not found.
 *If you are an authorized user contact the GPES system administrator. Goodbye now!

SYSTEM s 6 segment "t" is not found or you are not authorized to use this TBS. pel: msg.

Example:

*Beginning of attachment process.

**Enter the name of the TBS you wish to use now: xyz

SYSTEM s 6 segment "ctl_info.syscopy" is not found or you are not authorized to use this TBS. pel: entry not found.

**If you desire another TBS enter yes, otherwise no: yes

**Enter the name of the TBS you wish to use now:

User response:

If the user is an authorized user of the TBS, he should contact its system administrator.

SYSTEM s 7 routine "t" which calculates <object> "t1" is not found or you are not authorized to use it. pel: msg.

Example:

**COMMAND: calculate u(a);

SYSTEM s 7 routine aaa which calculates unit "a" is
not found or you are not authorized to use it. pel:

entry not found.

*Command ignored.

User response:

If he is an authorized user of the routine, it may be that
the routine is not in his search rules list. He should
add the directory containing the routine to his search
rules and try again.

Example:

**COMMAND: escape;

**multics command: asr...

(enter a blank line to return to system)

**COMMAND: calculate u(a);

**Entering routine aaa for level 1 calculation of unit "a".

**COMMAND:

SYSTEM s 8 routine "t" which evaluates pre-defined function "t1"
is not found or you are not authorized to use it. pel:

msg.

Example:

**COMMAND: sp v(x = f1(1,2)+4);

SYSTEM s 8 routine "fff" which evaluates pre-defined
function "f1" is not found or you are not authorized to
use it. pel: entry not found.

*Command ignored.

SYSTEM s 9 you are not authorized to use this TBS.

Example:

**Enter the name of the TBS you wish to use now: xyz

SYSTEM s 9 you are not authorized to use this TBS.

**If you desire another TBS enter yes, otherwise, no:

SYSTEM s 10 component public file "t" does not exist or you are not authorized to use it. cdbsys: msg.

Example:

**COMMAND: open c public file (ddd);

(user is prompted for the pathname of the file)

SYSTEM s 10 public file "ddd" does not exist or you are not authorized to use it. cdbsys: entry not found

SYSTEM s 11 this TBS does not exist or currently is not in operation.

Example:

**Enter the name of the TBS you wish to use now: xyz

SYSTEM s 11 this TBS does not exist or currently is not in operation.

SYSTEM s 12 you have created segment "t" outside of this program. Process cannot be saved under this same.
pdbsys: msg.

Example:

****COMMAND:** save pr test8;

*****SYSTEM***** s 12. You have created segment
"xtest81.GPES" outside of this program. Process cannot be
saved under this name.

*****SYSTEM***** s 13 a new segment cannot be created. pdbsys: msg.

Example:

****COMMAND:** save pr test12;

*****SYSTEM***** s 13 a new segment cannot be created.
pdbsys: msg.

*****SYSTEM***** s 14 invalid access for storing a process into the current
process file.

Example:

****COMMAND:** save pr test2;

*****SYSTEM***** s 14 invalid access for storing a process
into the current process file.

*****SYSTEM***** s 15 invalid access for deleting a process of the current
process file.

Example:

****COMMAND:** delete pr(d1,d2);

*****SYSTEM***** s 15 invalid access for deleting a process of
the current process file.

SYSTEM s 16 invalid pathname or no access for opening process
file "t".

Example:

**COMMAND: open pr file (xx);

(User is prompted for the pathname of the file)

SYSTEM s 16 invalid pathname or no access for
opening process file "xx".

SYSTEM s 17 you have deleted segment "t" outside of this program
or you may not have the proper access for this segment.
pdbsys: msg.

Example:

**COMMAND: load pr test;

SYSTEM s 17 you have deleted segment "xtest1.GPES"
outside of this program or you may not have the proper
access for this segment. pdbsys: entry not found.

SYSTEM s 18 invalid pathname or no access for opening component
file "t". cdbsys: msg.

Example:

**COMMAND: open c private file (x);

(User is prompted for the pathname of the file)

SYSTEM s 18 invalid pathname or no access for
opening component file "x". cdbsys: bad syntax in the
pathname.

SYSTEM s 19 invalid access for storing, deleting, copying, or including operation on component private file t.

Example:

**COMMAND: save c (c1, c3);

SYSTEM s 19 invalid access for storing, deleting, copying, or including operation on component private file x.

APPENDIX E

LITERATURE CITATIONS

1. Andrew, S.M. "Computer Flowsheeting Using Network 67: An Example," Trans. Inst. Chem. Eng., 46 (4), T123-T132 (1968).
2. Arab-Ismaili, M.S., "A Ph.D. Proposal for Design of an Interactive Computer System for Engineering of Chemical Processes," Massachusetts Institute of Technology (April 1975).
3. ASPEN, "Computer-Aided Industrial Process Design," First Quarterly Progress Report, MIT Report No. 2295T9-1 (August 1976).
4. ASPEN, "Computer-Aided Industrial Process Design," Second Quarterly Progress Report, MIT Report No. 2295T9-2 (November 1976).
5. ASPEN, "Computer-Aided Industrial Process Design," Third Quarterly Progress Report, MIT Report No. 2295T9-3 (February 1977).
6. ASPEN, "Computer-Aided Industrial Process Design," First Annual Report, MIT Report No. 2295T9-4 (June 1977).
7. ASPEN, "Computer-Aided Industrial Process Design," Fifth Quarterly Progress Report, MIT Report No. 2295T9-5 (September 1977).
8. ASPEN, "Computer-Aided Industrial Process Design," Sixth Quarterly Progress Report, MIT Report No. 2295T9-6 (December 1977).
9. ASPEN, "Computer-Aided Industrial Process Design," Appendix I to the Sixth Quarterly Progress Report, MIT Report No. 2295T9-7 (December 1977).
10. ASPEN, "Computer-Aided Industrial Process Design," Seventh Quarterly Progress Report, MIT Report No. 2295T9-8 (April 1978).
11. Barkley, R.W. and Motard, R.L., "Decomposition of Nets," Chem. Eng. J., 3 (3), 265 (1972).
12. Barnes, J.G.P., "Network 67: A General Description," Imperial Chemical Industries, Ltd., London, England (1967).
13. Batstone, D.B., Fenton, G. and Price, R.G.H., "The Steady State Digital Simulation of Chemical Plant of Arbitrary Configuration," Paper B2 Presented at IFAC Symposium of Digital Simulation of Continuous Processes, GYOR, Hungary (1971).
14. Briddell, E.T., "Process Design by Computer, Part 1," Chem. Eng., 81, (3) 60 (1974a).

15. Briddell, E.T., Part 2, Chem. Eng., 81, (5) 113 (1974b).
16. Briddell, E.T., Part 3, Chem. Eng., 81, (7) 77 (1974c).
17. Briggs, D.E., Carnahan, B. and Lopez, L.A., "DYSCO: An Interactive Executive Program for Dynamic Simulation and Control of Chemical Processes," Paper presented at the AIChE 78th Nat. Meeting, August 18-21, Salt Lake City (1974).
18. Cavett, R.H., "Flowtran Physical Properties," 49th NGPA Annual Convention, preprint, Natural Gas Producers' Association, Denver, Colorado (March 17-19, 1970).
19. Cavett, R.H., "Monsanto Physical Data System," AIChE 65th Annual Meeting, New York (1972).
20. Chueh, C.F. and Stein, T.W., "A Comprehensive Thermal and Physical Properties Information System-Halcon Physical Properties System," Chemical Engineering Computing, Vol. 2, AIChE Workshop Series (1972).
21. CONCEPT - Mark III, Computer Aided Design Center, Cambridge, U.K. (1973).
22. Crowe, C.M., Hamielec, A.E., Hoffman, T.W., Johnson, A.I., Shannon, P.T. and Woods, D.R., Chemical Plant Simulation, Prentice Hall, New Jersey (1971).
23. Davies, C. and Perris, F.A., "Experience in the Industrial Application of Generalized and Special-Purpose Computer Programs for the Steady-State Design and Stimulation of Complete Chemical Processes," Paper D1 presented at IFAC Symposium of Digital Simulation of Continuous Processes, GYOR, Hungary (1971).
24. Debrosse, C.J. and Westerberg, A.W., "A Feasible Point Algorithm for Structured Design Systems in Chemical Engineering," AIChE J., 19, (2) 251 (1973).
25. Dodrill, W.H., "Using GIFS in the Analysis and Design of Process Systems," Proceedings of the Fall Joint Computer Conference, Vol. 22, pp. 275-279, American Federation of Information Processing Services, Philadelphia, Pa. (1962).
26. Donovan, J.J., Systems Programming, McGraw-Hill, New York (1972).
27. Edie, F.C. and Esterberg, A.W., "Computer Aided Design, Part 3, Decision Variable Selection to Avoid Hidden Singularities in Resulting Recycle Calculation," Chem. Eng. J., 2, 114 (1971).

28. Elzy, E., "DISCOSSA," Department of Chemical Engineering, Oregon State University, Corvallis, Ore. (1969).
29. Evans, L.B., "Chemical Process Systems Analysis," a set of unpublished class notes, Massachusetts Institute of Technology (1975).
30. Evans, L.B., Joseph, B., and Seider, W.D., "Computer Aided Industrial Process Simulation and Design," Paper presented at the Conference on Mathematical Modeling of Coal Conversion Processes, Washington, D.C., November 1976.
31. Evans, L.B., and Seider, W.D., "The Proposal to Develop IEPES: Response from Industry," Report (January 22, 1976).
32. Evans, L.B., Joseph, B., and Seider, W.D., "Systems Structures for Process Simulation," AIChE Journal, 23 (5) 658 (1977).
33. Evans, L.B., and Seider, W.D., "The Requirements for an Advanced Computing System," CEP, 72 (6) (1976).
34. Evans, L.B., Steward, D.G. and Sprague, C.R., "Computer Aided Chemical Process Design," Chem. Eng. Progr., 64 (4), 39-46 (1968).
35. Fay, J.E., "A Prototype System for On-Line Computer-Aided Process Design of Heat Exchange Networks," Sc.D Thesis, Massachusetts Institute of Technology, Cambridge, Mass. (1971).
36. Flower, J.R. and Whitehead, B.D., "Computer-Aided Design: A Survey of Flowsheeting Programs. Part 1," Chem. Eng., (London), 272, 208 (1973a).
37. Flower, J.R. and Whitehead, B.D., "Computer-Aided Design: A Survey of Flowsheeting Programs. Part 2," Chem. Eng., (London), 273, 271 (1973b).
38. FLOWPACK - User's Manual, 1970 (Runcorn; I.C.I. Ltd., CIRL).
39. Forder, G.J., and Hutchinson, H.P., "The Analysis of Chemical Plant Flowsheets," Chem. Eng. Sci., 24, 771-785 (1969).
40. Franks, R.G.E., Modeling and Simulation in Chemical Engineering, Wiley-Interscience, New York (1972).
41. Gaddy, J.L., "The Use of Flowsheet Simulation Programs in Teaching Chemical Engineering Design," Chem. Eng. Ed., 8 (3), 124 (1974).

42. Garcia-Gamboa, E., "Estimation of Physical Properties for Coal-Derived Liquids," M.S. Thesis, Chem. Eng. Dept, MIT (1977).
43. General Electric Company, "User's Guide, Process Design System -- GEPDS (Version 2)," Information Services Department, Bethesda, Md. (1970).
44. Goodson, P.D., "The Applicability of Flowtran to Coal Conversion Process Analysis," M.S. Thesis, University of Wisconsin (1976).
45. Gray, G.C., "Compound Data Structure for Computer-Aided Design: A Survey," Proceedings of 22nd ACM National Meeting, Vol. 22, pp. 355-367, Association for Computing Machinery, Washington, D.C. (1967).
46. Ham, P.G., "The Transient Analysis of Integrated Chemical Processes," Ph.D. Dissertation, University of Pennsylvania, Philadelphia (1971).
47. Hankinson R.W. and Cantwell, K.R., "The Phillips Petroleum Company Computer Stored Physical Data System," Chemical Engineering Computing, Vol. 2, AIChE Workshop Series (1972).
48. Hanyak, M.E. and Kalus, R.L., "Use of Translator Writing Systems in the Development of Problem Oriented Languages," Paper presented at AIChE 66th Annual Meeting, Philadelphia, Pa. (1973).
49. Henley, E.J. and Rosen, E.M., Material and Energy Balance Computations, Wiley, New York (1969).
50. Himmelblau, D.M., Basic Principles and Calculations in Chem Eng, 3rd Ed., Prentice-Hall (1974).
51. Himmelblau, D.M., and Bischoff, K.B., Process Analysis and Simulation, John Wiley (1968).
52. Holland, C.D., Multicomponent Distillation, Prentice-Hall, Englewood Cliffs, New Jersey (1968).
53. Honeywell Information Systems Inc., MULTICS PL/1 Language Specifications, Order No. AG94 (1976).
54. Honeywell Information Systems Inc., MULTICS PL/1 Reference Manual, Order No. AM83 (1976).
55. Honeywell Information Systems Inc., MULTICS Programmers' Manual, Commands and Active Functions, Order No. AG92 (1976).

56. Honeywell Information Systems Inc., MULTICS Programmers' Manual, Introduction, Order No. AG90 (1973).
57. Honeywell Information Systems Inc., MULTICS Programmers' Manual, Reference Guide, Order No. AG91 (1975).
58. Honeywell Information Systems Inc., MULTICS Programmers' Manual, Subroutines, Order No. AG93 (1976).
59. Honeywell Information Systems Inc., MULTICS Programmers' Manual, Subsystem Writers' Guide, Order No. AK92 (1977).
60. Hughes, R.R., Singer, E. and Souders, M., "Machine Design of Refineries," Sixth World Petroleum Congress, Section VII, Paper 17, pp. 93-102, Frankfurt, W. Ger. (1963).
61. Hughson, R.V. and Steymann, E.H., "Computer Programs for Chemical Engineers," Chem. Eng., 78 (14), 66 (1971).
62. Hughson, R.V. and Steymann, E.H., "Computer Programs for Chemical Engineers, 1973, Part 1" Chem. Eng., 80 (19), 121 (1973a).
63. Hughson, R.V. and Steymann, E.H., "Computer Programs for Chemical Engineers, 1973, Part 2" Chem. Eng., 80 (21), 127 (1973b).
64. Ingels, D.M., "A System for Simulating Chemical Process Dynamics and Control," Ph.D Dissertation, University of Houston (1970).
65. International Business Machines Corporation, "IBM System/360 Operating System PL/1(F) Language Reference Manual," Form GC28-8201-4, IBM Corp., White Plains, New York (1972).
66. International Business Machines Corporation, "IBM System/360 Operating System PL/1(F) Programmer's Guide," Form GC28-6594, IBM Corp., White Plains, New York (1972).
67. International Business Machines Corporation, "Problem Language Analyzer (PLAN): Program Description Manual" Technical Publications Department, White Plains, New York (1969).
68. Jain, Y.V.S. and Eakman, J.M., "Identification of Process Flow Networks," Paper presented at AIChE 69th National Meeting, Houston, Texas (1971).
69. James, J.L., Gardner, N.F., Reinhart, L.R. and Hellenack, L.J., "Economic Design by Flexible Flowsheet Analysis," I. Chem. E. Symposium Series, Vol 18., pp. 11-18, Institution of Chemical Engineers, London, England (1966).

70. Johnson, A.I., "The Modular Approach Applied to the Unsteady State Behavior of Systems," Brit. Chem. Eng., Proc. Tech., 17 (3) 217 (1972a).
71. Johnson, A.I., "Computer Aided Process Analysis and Design - A Modular Approach," Brit. Chem. Eng. Proc. Tech., 17 (1) 28 (1972b).
72. Johnson, A.I., and Toong, T., "The Modular Approach to System Analysis and Design," (GEMCS Manual), McMaster University (1968).
73. Johnson, A.I., and Toong, T., "GEMCS General Electric/Mcmaster Simulator: Information Handling Program for Analysis and Design of Engineering and Management Systems," Department of Chemical Engineering, McMaster University, Hamilton, Ont., and Canadian General Electric Company, Toronto, Ont. (1968).
74. Joseph, B., Evans, L.B. and Seider, W.D., "The Use of PLEX DATA Structure in Process Simulation," Computers and Chemical Engineering, Pergamon Press, to be published.
75. Kehat, E. and Shacham, M., "Chemical Process Simulation Programs-1," Process Technol., 18 (1/2), 35 (1973a).
76. Kehat, E. and Shacham, M., "Chemical Process Simulation Programs-2, Partitioning and Tearing System Flow Sheets" Process Technol., 18 (3), 115 (1973b).
77. Kehat, E. and Shacham, M., "Chemical Process Simulation Programs-3, Solution System of Nonlinear Equations" Process Technol., 18 (4/5), 181 (1973c).
78. Kenny, L.N. and Prados, J.W., "A Generalized Digital Computer Program for Performing Process Material and Energy Balances," University of Tennessee, Knoxville, Tenn. (1966).
79. Kesler, M.G. and Griffiths, P.R., "A Computer System for Process Simulation," Proceedings of the American Petroleum Institute, Section III, Vol. 43, pp. 49-56 (1963).
80. Kesler, M.G. and Kessler, M.M., World Petrol., 29, 60 (1958).
81. Kevorkian, A.K. and Snoek, J., "Decomposition of Large Scale Systems, Theory and Applications in Solving Large Sets of Non-Linear Simulations Equations," in Himmelblau, D.M. (ed), Decomposition of Large Scale Problems, North Holland/American Elsevier, Amsterdam (1973).

82. King, C.J., Foss, A.S., Grens, E.A., Lynn, S. and Rudd, D.F., "Chemical Process Design and Engineering," Chem. Eng. Ed., 7 (2), 72 (1973).
83. Kliesch, H.C., "An Analysis of Steady-State Process Simulation: Formulation and Convergence," Ph.D. Thesis, Tulane University, New Orleans, La. (1967).
84. Klumpar, I.V., "Process Economics by Computer," Chem. Eng., 77 (1), 107 (1970a).
85. Klumpar, I.V., "Process Economics by Computer," Chem. Eng., 77 (14), 76 (1970b).
86. Kwon, Y.J., "Asymptotic Convergence Technique and Executive Concept for Steady-State Process Systems Analysis," Ph.D. Thesis, Oregon State University, Corvallis, Ore. (1969).
87. Lederman, P.B., "Flowsheet Simulation and Beyond," Chem. Eng., 75 (25), 127-132 (1968).
88. Leesley, M.E., "Process Plant Design by Computer," Process Technol., 18, (11) 403 (1973).
89. Loibl, J.M., Camp, D.T. and Wilkins, G.S., "The Dynamic Analysis of Chemical Processes with a User-Oriented Executive Program," Proc. 1973 Summer Computer Simulation Conference, July 17-19, Montreal Canada (1973).
90. Louis, J.F. et al., "Open Cycle Coal Burning MHD Power Generation, an Assessment and a Plan for Action," R&D Report No. 64, U.S. Department of Interior, Office of Coal Research (February 1972).
91. Louis, J.F. et al., "Open Cycle Coal Fired MHD Generation," MIT-ERDA Report No. 1209-1 (July 1975).
92. Louis, J.F. et al., "Critical Contributions in MHD Power Generation," MIT-ERDA Report No. 2215-1 (September 1975).
93. Louis, J.F. et al., "Critical Contributions in MHD Power Generation," MIT-ERDA Report No. 2215-2 (December 1975).
94. Louis, J.F. et al., "Critical Contributions in MHD Power Generation," MIT-ERDA Report No. 2215-3 (March 1976).
95. Louis, J.F. et al., "Critical Contributions in MHD Power Generation," MIT-ERDA Report No. 2215-4 (July 1976).
96. Louis, J.F. et al., "Critical Contributions in MHD Power Generation," MIT-ERDA Report No. 2215-5 (September 1976).

97. Louis, J.F. et al., "Critical Contributions in MHD Power Generation," MIT-ERDA Report No. 2215-6 (December 1976).
98. Louis, J.F. et al., "Critical Contributions in MHD Power Generation," MIT-ERDA Report No. 2215-7 (March 1977).
99. Louis, J.F. et al., "Critical Contributions in MHD Power Generation," MIT-ERDA Report No. 2215-8 (June 1977).
100. Louis, J.F. et al., "Critical Contributions in MHD Power Generation," MIT-ERDA Report No. 2215-9 (October 1977).
101. Louis, J.F. et al., "Critical Contributions in MHD Power Generation," MIT-U.S. Department of Energy Report No. 2215-10 (December 1977).
102. Louis, J.F. et al., "Critical Contributions in MHD Power Generation," MIT-U.S. Department of Energy Report No. 2215-11 (March 1978).
103. Macable, W.L. and Smith, J.C., Unit Operations in Chemical Engineering, McGraw-Hill, New York (1967).
104. Madnick, S.E. and Donovan, J.J., Operating Systems, McGraw-Hill, New York (1974).
105. Maejima, T., "Computer-Aided Chemical Process Design," M.S. Thesis, Massachusetts Institute of Technology, Cambridge, Mass. (1970).
106. Maejima, T. and Shindo, A., "System Structure of a Computer Aided System for Process Engineering," Proc. 1973 Summer Computer Simulation Conference, July 17-19, Montreal, Canada (1973).
107. Mah, R.S.H., "Recent Development in Process Design," Symposium on Basic Questions of Design Theory, Columbia Univ., (May 30-31, 1974).
108. Mah, R.S.H., "Structure Decomposition in Chemical Engineering Computation, AIChE 72nd National Meeting, St. Louis, Mo. (1972).
109. Mah, R.S.H. and Rafal, M., "Automatic Program Generation in Chemical Engineering Computation, Trans. Instn. Chem. Eng. 49, 407 (1971).
110. Meadows, E.L., Jr., "AIChE Physical Properties Project," Proceedings of the American Petroleum Institute, Section III, Vol. 44, pp. 300-303 (1964).

111. Meadows, E.L., Jr., "Estimating Physical Properties: The AIChE System," Chem. Eng. Progr., 61 (5), 93-95 (1965).
112. Moore, J.F., Bonner, J.S., and Karvelas, L.P., "Unit Operations Simulator," Bonner and Moore Engineering Associates, Houston, Tex. (1960)
113. Mosler, H.A., "PACER -- A Digital Computer Executive Program for Process Simulation and Design," M.S. Thesis, Purdue University, Lafayette, Ind. (1964).
114. Motard, R.L., "Optimization of Natural Gasoline Plant Operation," in "Computers in Engineering Design Education," Vol. II, pp. 36-70, University of Michigan, Ann Arbor, Mich. (1966).
115. Motard, R.L., Lee, H.M., Barkely, R.W., and Ingels, D.M., "CHESS, Chemical Engineering Simulating System, System Guide," Technical Publications Co., Houston, Tex. (1968).
116. Motard, R.L., Lee, H.M., "CHESS, Chemical Engineering User's Guide," 3rd Ed., University of Houston (1971).
117. Motard, R.L., Shacham, M., and Rosen, E.M., "Steady State Chemical Process Simulation," AIChE J., 21 (3), pp. 417-436 (1975).
118. Myers, A.L. and Seider, W.D., Introduction to Chemical Engineering and Computer Calculations, Prentice-Hall, New Jersey, 1976.
119. Nagiev, M.F., "Material Balance in Complex and Multistage Recycle Chemical Processes," Chem. Eng. Progr. 53, 297-303 (1957).
120. Naphtali, L.M., "Process Heat and Material Balance," Chem. Eng. Progr., 60 (9), 70-74 (1964).
121. Newman, W.M., "A System for Interactive Graphical Programming," Proceedings 1968 Spring Joint Computer Conference, Vol. 33, pp. 47-54, American Federation of Information Processing Societies, Washington, D.C. (1968).
122. Norris, R.C., "Estimating Physical Properties - Route Selection," CEP, 61, 5, 96-101 (May 1965).
123. Nott, H.D., "SLED: Simplified Language for Engineering Design - A Computerized System for the Design of Chemical Processes," Ph.D. Dissertation, University of Michigan (1971).
124. Nuttal, H.E., Jr. and Himmelblau, D.M., "Interactive Reactor Simulation," Proc. 1973 Summer Computer Simulation Conference, July 17-19, Montreal, Canada.

125. PACER 245 User Manual, Hanover, New Hampshire: Digital Systems Corporation (1971).
126. Peiser, A.M. and Kessler, M.M., "The Computer Approach to Optimizing Plant Design," Refining Eng., 32 C7-C9 (1960).
127. Perry, R.H., Chemical Engineers' Handbook, 5th Edition, McGraw-Hill, New York (1973).
128. Peters, N. and Barker, P.E., "PEETPACK - A Non-Proprietary Flowsheeting Program," Chem. Engr., (London), pp. 763-768, (December 1974).
129. Peters, N. and Barker, P.E., "An Appraisal of the Use of PACER, GEMCS and CONCEPT for Chemical Plant Simulation and Design," Chemical Eng., (London), 283, 149 (1974).
130. Peterson, J.N., "Survey of Software for Computer-Aided Chemical Process Design," M.S. Thesis, Department of Chemical Engineering, MIT, January 1977.
131. Petroleum Consultants, "An Introduction to the Chem. E. Simulator," Petroleum Consultants, Houston, Tex.
132. Pho, T.K. and Lapidus, L., "Topics in Computer Aided Design; Part 1 An Optimum Tearing Algorithm for Recycle Systems," AIChE J., 19 (6), 1170 (1973).
133. Porter, J.H., "Computer-Aided Process Design: An Exercise in Dynamic Man-Machine Communication," Paper presented at the Annual Meeting of AIChE, Los Angeles (December 1969).
134. Powers, G.J., "Heuristic Synthesis in Process Development," Chem. Eng. Prog., 68 (8) (1972).
135. Poznanovic, D.S., and Seider, W.D., "A Physical Property Information System for Undergraduate Education," Chemical Engineering Computing, Vol. 1, AIChE Workshop Series (1972).
136. PPDS - Physical Property Data System, The Institution of Chem. Eng., London (1971).
137. Ramirez, W.F. and Vestal, C.R., "Algorithms for Structuring Design Calculations," Chem. Eng. Sci., 27, 2243 (1972).
138. Ravicz, A.E., and Norman, R.L., "Heat and Mass Balancing on a Digital Computer," Chem. Eng. Progr., 60 (5), 71-76 (1964).

139. Reid, R.C. and Evans, L.B., "Design Data for Industry, Property Prediction with Computer System," AIChE Today Series, AIChE, New York (1970).
140. Reid, R.C., Prausnitz, J.M. and Sherwood, T.K., The Properties of Gases and Liquids, 3rd Ed., McGraw-Hill, New York (1977).
141. Rinard, I.H. and Ripps, D.L., "The Steady State Simulation of Continuous Chemical Processes," Chemical Engineering Progress Symposium Series No. 55, Vol. 61, pp. 34-51 (1965).
142. Ross, D.T., The AED Approach to Generalized Computer-Aided Design, Proceedings A.C.M. National Meeting (1967).
143. Rubin, D.I., "Generalized Material Balance," Chemical Engineering Progress Symposium Series No. 37, Vol. 58, pp. 54-61 (1962).
144. Rudd, D.F., "The Synthesis of System Designs: I. Elementary Decomposition Theory," A.I.Ch.E. Journal, 14 (2), 343-349 (1968).
145. Rudd, D.F. and Watson, C.C., Strategy of Process Engineering, Wiley (1968).
146. Sargent, R.W.H., "Integrated Design and Optimization of Processes," Chem. Eng. Progr., 63 (9), 71-78, (1967).
147. Sargent, R.W.H., "Developments in Computer-Aided Process Design, The Chem. Eng., (224), CE424-CE427 (1968).
148. Sargent, R.W.H. and Westerberg, A.W., "'Speed-Up' in Chemical Engineering Design," Trans. Inst. Chem. Eng., 42, T190-T197 (1964).
149. Seader, J.D., Seider, W.D., and Pauls, A.C., "FLOWTRAN Simulation: An Introduction - CACHE Committee," Ulrich's Bookstore, Ann Arbor, Michigan (1974).
150. Seider, W.D. (MOD), "Computer Aided Analysis and Design Packages," Chemical Engineering Computing, Vol. 2, AIChE Workshop Series (1972).
151. Seider, W.D., Evans, L.B., Joseph, B., Wong, E., "Routing in Process Simulation," Paper presented at the 69th Annual Meeting of the AIChE, Chicago, Illinois (November 1976).
152. Service Bureau Corporation, "GIFS -- Generalized Interrelated Flow Simulation User's Manual," Service Bureau Corporation, New York, N.Y. (1962).

153. Service Bureau Corporation, "Chemical Engineering Information Processing System -- CHIPS 2 -- User's Manual," Computing Services Division, New York, N.Y. (1968).
154. Shannon, P.T. and Frantz, D.R., "The PACER System Manual," Dartmouth College, Hanover, N.H. (1966).
155. Shannon, P.T., Johnson, A.I., Crowe, C.M., Hoffman, T.W., Hamielec, A.E., and Woods, D.R., "Computer Simulation of a Contact Sulfuric Acid Plant," Chem. Eng. Progr., 62 (6), 49-59 (1966).
156. Smith, B.D., Design of Equilibrium and Stage Processes, McGraw-Hill (1963).
157. Soylemez, S. and Seider, W.D., "The Chemical Engineering-Process Analysis and Design Interface," Chemical Engineering Computing, Vol. 2, AIChE Workshop Series (1972).
158. Steward, D.G., "A Survey of Computer-Aided Chemical Process Design," M.S. Thesis, Massachusetts Institute of Technology, Cambridge, Mass. (1967).
159. Tsubaki, M., "Computer Aided Process Analysis and Design," Chem. Eng. Progr., 69 (9), 78 (1973).
160. Uhde, Freidrich, Thermophysical Properties Program Package, GMBH, Dortmund, Germany (1973).
161. Vora, K.T., "The Template Data Base System and Process Module Interface Design for the Interactive Chemical Process Engineering System," M.S. Thesis, MIT, Cambridge, Mass. (June 1977).
162. Wells, G.L. and Robson, P.M., Computation for Process Engineers, Wiley, New York (1973).
163. Westerberg, A.W., "Decomposition Methods for Solving Steady State Process Design Problems," p. 379 in Himmelblau, D.M. (ed), Decomposition of Large Scale Problems, North Holland/American Elsevier, Amsterdam (1973).
164. Westerberg, A.W., "generalized Symbolic Programme for the Analysis of Interlinked Chemical Engineering Processes," Ph.D. Thesis, University of London, London, England (1964).
165. White, G.L., "Steady-State Simulation of the Hygas Coal Gasification Process," M.S. Thesis, Chem. Eng. Dept., MIT (1977).
166. Wong, E., "Routing of Estimation Methods for Calculating Physical Properties," M.S. Thesis, MIT, Cambridge (June 1976).

167. Worley, F.L., Jr. and Motard, R.L., "Information Systems in Chemical Engineering Design," Chemical Engineering Computation, Vol. 2, AIChE Workshop Series (1972).
168. Worley, F.L., Jr. and Motard, R.D., CEDA - Control Equipment Design and Analysis, University of Houston, Texas (1976).

BIOGRAPHICAL SKETCH

Mohammad Sharif Arab-Ismaili was born on February 10, 1949 in Shahrood, Iran, the son of Ali-Mohammad and Jalileh Arab-Ismaili. He received his formal education in the public school system of that city. The author then entered Abadan Institute of Technology, majoring in Chemical Engineering. He earned a B.S. with honors in September 1971, and was class valedictorian. While attending A.I.T., he held summer jobs at the Abadan Oil Refinery, the B.F. Goodrich Rubber Company, the Abadan Petrochemical Company, and the Computer Information Services of the Iranian Oil Operating Companies. Upon graduation he spent two years with the Iranian Oil Operating Companies as a computer systems analyst.

Entering the Massachusetts Institute of Technology in September 1973, the author began work toward his Ph.D. in the Department of Chemical Engineering. On December 20, 1976 he married Mitra Khademi, and they have an eight-month-old daughter, Neda Arab-Ismaili.

The author's immediate plan is to complete his studies toward the degree of Master of Science in Management at MIT's Sloan School of Management.

May 1978