Policy Analysis Language, version 2.3: programmer's guide for CDC Cyber computers

Winer, Claudia; Wolf, Christopher

1976

66p.

Mich. State

(In Computer Library for Agr. Systems Simulation. CLASS doc.no.3)

Computer programs Simulation
Data processing
Decision making
Programming languages
# TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>CHAPTER</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>I. INTRODUCTION</td>
<td>1</td>
</tr>
<tr>
<td>II. THE DECLARATIVE SECTIONS</td>
<td>6</td>
</tr>
<tr>
<td>III. THE CONVERSATION SECTIONS</td>
<td>11</td>
</tr>
<tr>
<td>IV. COMMAND USAGE</td>
<td>19</td>
</tr>
<tr>
<td>V. THE STANDARD VALUE FILE</td>
<td>28</td>
</tr>
<tr>
<td>VI. DEBUGGING A PAL PROGRAM</td>
<td>36</td>
</tr>
<tr>
<td>VII. PREPARING A MODEL FOR USE WITH A PAL CONVERSATION</td>
<td>42</td>
</tr>
<tr>
<td>VIII. INSERTING FORTRAN IN A PAL PROGRAM</td>
<td>45</td>
</tr>
<tr>
<td>IX. OVERLAYING A PAL PROGRAM</td>
<td>49</td>
</tr>
<tr>
<td>APPENDIX A - USING PAL ON THE CDC 6500</td>
<td>56</td>
</tr>
<tr>
<td>APPENDIX B - FORTRAN CODE GENERATED BY THE PAL TRANSLATOR</td>
<td>59</td>
</tr>
</tbody>
</table>
The Computer Library for Agricultural Systems Simulation (CLASS) is one of the four major activities of the Agricultural Sector Analysis and Simulation Projects at Michigan State University under U.S. Agency for International Development Contract AID/csd-2975. The other three major interrelated project activities include theoretical and methodological research, the Development Analysis Study Program, and field activities, primarily in the Republic of Korea.

The project objective is to develop an approach to institutionalizing an analytical capacity for planning, policy formulation, program development, and project implementation for agricultural sector development within the public decision making structure of developing countries. A major component of the analytical capacity is a series of system simulation models tailored to the needs of the individual country. Much of the experience gained from the field activity and the knowledge gained from the theoretical and methodological research added to the present stock of knowledge about building and maintaining analytical capacities for agricultural sector development can be preserved and extended in the training provided through the Development Analysis Study Program and in the stock of model, component, and utility routine computer software documented in the Computer Library for Agricultural Systems Simulation.

In full operation, the Computer Library for Agricultural Systems Simulation (CLASS) acquires, catalogs, maintains and distributes computer programs and associated documentation. These computer programs are of generalized simulation models, components, and routines designed specifically for the analysis of agricultural development problems and processes. In particular, the library sets standards of admissibility for programs and documentation; catalogs and indexes programs and documentation so as to facilitate their retrieval by users seeking a set of programs to be used in a specific problem analysis; and distributes programs and documentation to users.

To enhance the effectiveness of the library, its functions also include identifying and soliciting needed models; actively bringing programs and documentation up to the library's standards; and providing limited consultation in identifying and implementing appropriate library programs for a particular application. A subsidiary function of the library in conjunction with the identification and solicitation of models is to survey and catalog ongoing research in agricultural systems modeling and simulation.

The CLASS document publication series is the main vehicle for informing potential users of the substance of CLASS holdings and activities.

July, 1976
George E. Rossmiller
Director
Agricultural Sector Analysis and Simulation Projects
PREFACE

This document is intended as a guide for the programmer who is using the Policy Analysis Language (PAL) Version 2.3. It explains the use of PAL, from writing a program to running it. There are two related documents. The PAL Reference Manual (CLASS-4) defines the syntactical elements of the language. Guide to Using A PAL Program (CLASS-5) is a short guide written for nonprogrammers who will be using a PAL program from a terminal.

The initial programming for PAL was done by Frank Huybrechts, Carl Wright, and Tom Hamby. We wish to thank Dave Watt for his help in many areas, especially in testing PAL; Mort Rahimi and Mike Abkin for the initial conceptualization and valuable design suggestions; and Judy Pardee for her infinite patience throughout the numerous retypings of this manual.

Documentation and consultation are available free of charge. Source code for the translator and the library routines can also be provided. For more information, or for comments on PAL or this manual, contact Chris Wolf or Claudia Winer in Room 306 Computer Center (355-4701).

Chris Wolf
Claudia Winer
CHAPTER I
INTRODUCTION

Purpose

The Policy Analysis Language (PAL) was developed to make computer simulation models easy for nontechnical people to use. Normally when such a user wants to make test runs of a model, he needs the help of a programmer or modeler or both. The modeler must explain to him the model structure, standard input and output options, etc. The programmer must then make the necessary changes, run the program, and give the results to the user.

PAL can simplify this process by allowing the nonprogrammer to run the model himself. The modeler can write a PAL program which will conduct a conversation with the user at the terminal. The program can explain to the user the same things the model designer might have explained, such as the model structure and the areas that can be explored. This program can allow the user to change variables which are of interest to him, to choose the output he desires, and to make repeated runs of the model.

A PAL program may be constructed to offer the user two alternative sets of messages—typically a set of detailed explanations for the novice user and a set of brief prompts for the experienced user. In addition, whenever the user is expected to enter input, he may choose to enter commands for changing and printing data values, running the model, or performing various other actions. If desired, all functions may be performed by commands, bypassing the conversation entirely. This capability is particularly useful to the programmer when checking out the model.
Although developed for use with simulation models, PAL is suitable for use with almost any FORTRAN program. Throughout this manual, we will use the term "model" to refer to the FORTRAN program.

**Structural Overview**

A PAL program consists largely of text statements which are printed at the terminal at execution time. There are also statements which read in the user's responses to the text and perform various functions depending on his responses. If the user desires, he may bypass the pre-programmed conversation and enter commands directly from the terminal to do what he wants.

The major divisions of a PAL program are called sections. The structure of a PAL program is defined by ten PAL directives which specify the sections of the program. Each directive name begins with a dollar sign ($). The directives are classified into three types as shown below:

<table>
<thead>
<tr>
<th>Declarative Section Directives</th>
<th>Conversation Section Directives</th>
<th>Language Directives</th>
</tr>
</thead>
<tbody>
<tr>
<td>$MODEL</td>
<td>$PROGRAM</td>
<td>$FORTRAN</td>
</tr>
<tr>
<td>$COMMON</td>
<td>$SUBROUTINE</td>
<td>$PAL</td>
</tr>
<tr>
<td>$LOCAL</td>
<td>$END</td>
<td></td>
</tr>
<tr>
<td>$TYPE</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$CONSTRAINTS</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The $MODEL section tells the name of the model to be used with the conversation. The $COMMON section is used for declaring the variables to be shared between the PAL program and the model. The $LOCAL section is used for declaring variables to be used only in the PAL program. In the $TYPE section, variables whose types are to be different than their FORTRAN default can have their types specified. The $CONSTRAINTS section
is used by the model designer to disallow particular variable values that he feels are unreasonable or could cause errors in the model. The $PROGRAM$ and $SUBROUTINE$ sections contain the text for the conversation and the program for executing it. In addition, actual FORTRAN statements may be inserted by the use of $FORTRAN$ and $PAL$ directives. None of these sections is required, but most programs will contain at least a $MODEL$ section, $COMMON$ section, and $PROGRAM$ section.

**Using a PAL Program**

There are two steps required in compiling a PAL program. First you use the PAL translator, which reads in a PAL program and translates it into FORTRAN. Then you compile this FORTRAN program, using the FORTRAN compiler. The result of this will be a binary version of the conversation. See Appendix B for more detailed information about the contents of this file.

You must also create a file which contains default values for the variables used in the PAL program. This is called the standard value file. Chapter V has more information about the standard value file.

To execute the program you need four files: the conversation file, the standard value file, the file containing the compiled model, and a file containing the PAL library routines (which carry out most of the functions of PAL). These files are depicted in Figure 1.

The conversation file will contain a main program which will be the overall "executive" for the run. The model will consist of one or more subroutines. To execute the model, the conversation will call the "executive" subroutine of the model. In some models this will be the only
Figure 1
subroutine; in others, this routine will call other routines which will carry out various functions. The model will print the results for the user and, when it is finished, control will return to the conversation. The conversation may end at that point or it may allow the user to make more changes and make further runs. The user can end the conversation whenever he wishes.

Appendix A explains in more detail the procedures for compiling and running a PAL program.
CHAPTER II
THE DECLARATIVE SECTIONS

$MODEL Section

The $MODEL section consists of one card which gives the name of the model. When the model is used with the conversation, the main program of the model must be converted to a subroutine. When the time comes in the conversation to execute the model, the subroutine whose name is on the $MODEL card will be called.

$COMMON Section

A $COMMON section is a means of declaring that a FORTRAN common block from the model is to be used in the PAL program. The $COMMON section will contain all of the variables from the common block, with their dimensions. Each variable may also have an "explanation." The explanation is what will be shown to the user if he requests further information about that variable. The explanation should be written in nontechnical terms that the user can readily understand. It should not be thought of as program documentation.

The variables in a $COMMON section are used to create a labeled common block. The variables will appear in the common block in the order in which they occur in the $COMMON section. Normally, the variables will match those in the model's common block in name, type, and dimensions. If they are not identical, care should be taken to insure that the variables are aligned properly and that the total sizes of the blocks are the same.

Any model variables that are to be referenced in the PAL program must be included in the $COMMON sections. It is the only way of communicating
between the model and the PAL program, so all policy variables should be included. Also, any variable whose value the user may want to change should be included, as well as any variable whose value is to be printed by the PAL program.

Standard and Current Values

A simulation model will usually have a "standard" or "base-run" value assigned to each variable in the program. When the model is run using these values, it is considered a "base run." For testing purposes, one or more values may be changed and the results compared with those of the base run. These values can also be used as default values, so the user only changes those he is interested in and the rest retain their standard values automatically. In order to run any PAL program which contains one or more $COMMON sections, you must provide a file of standard values.

The value that a variable has at any given time is referred to as its "current" value. When a PAL program begins executing, all common variables are initialized to their standard values so that the current and standard values are the same. Whenever a value is changed by the PAL program, the model, or the user, only the current value changes.

For more information about the standard value file see Chapter V.

$LOCAL Section

The $LOCAL section is a means of declaring the variables that are to be used only in the PAL program and not in the model. Variable explanations, as provided for in the $COMMON section, are not allowed in the $LOCAL section. The standard value file, which contains default values for all those variables which occur in the $COMMON sections, does
not have values for local variables. However, the local variable statement may specify initial data values in a way similar to a FORTRAN DATA statement.

$\textbf{TYPE Section}$

The $\textbf{TYPE}$ section is used to declare nonstandard variable types. A variable will automatically assume the type it would have as a FORTRAN variable. If you wish to change the default type, a REAL or INTEGER statement may be used. There are also two non-FORTRAN variable types: HOLLERITH and INITIAL.

A HOLLERITH variable is one that is used for character information. REAL and INTEGER variables should not be used for character information, and HOLLERITH variables should not be used for numeric information.

A type INITIAL variable may be declared to be any of the other three types as well. Type INITIAL is particularly useful for simulation models. It is used for a variable whose value changes during a simulation run and therefore must be re-initialized for the next run. Since LOCAL variables are used only in the PAL program—not in the model—they cannot change during a run. For this reason, LOCAL variables should not be declared type INITIAL.

As an example, suppose a user wishes to make several runs of a simulation model of Michigan. The price of horses is a policy variable. Its standard value is $700 and the price increases 10% per year for the length of the run. The user follows the conversation, changing several variables. He decides that the initial value for the price of horses will be $750 for this set of runs. After he makes these changes, he is
ready to run the model. When the model is run, the price of horses will increase from $750 to $1098 (for a five-year run).

The user will then be asked if he wishes to make more changes and do another run. If he wants the next run to incorporate all of the changes from the previous run, he has a problem. If he leaves all the values as they are, the price of horses will start at $1098 for the next run. If all of the variables are set back to their default values, the price of horses will start at $700 instead of $750, and he will lose all the other changes he made as well.

By declaring the price of horses to be type INITIAL, this problem can be resolved. There is a command called >STORIV which can be executed, just before a run of the model is made, to save the current values of all type INITIAL variables. Then, after the run is made, those values can be restored with a command called >INITIALIZE. The values of non-INITIAL variables will not be affected by either of these two commands. The price of horses will be $750 and the user can make whatever other changes he wants for the next run. See Chapter IV for more information on the use of >STORIV and >INITIALIZE.

$CONSTRAINTS Section

The $CONSTRAINTS section contains statements which restrict the values that variables may take on. Any of the variables in the $COMMON or $LOCAL sections may be used. This gives the model designer a means of disallowing values that are incorrect or unreasonable. Anything which might cause a program error (such as dividing by zero) should also be included.

After the user has entered all of his values, they can be checked against the constraints. If any of the constraints are violated, they
will be printed at the terminal, and the user will be given instructions on how to fix them. The constraints that are violated are printed exactly as they are written—as FORTRAN logical expressions. They should be as readable as possible because constraint violations will not be an easy thing for a user to fix. He will have to understand the FORTRAN expression, and he will have to be able to enter the necessary PAL commands to fix them.

In some instances, it might be easier for the user if the checking were done as he enters the values. Then he can be given more specific feedback on what he has done wrong as soon as he has done it. The >REQUEST command is sometimes useful for this purpose. See Chapter IV for more information.
CHAPTER III
THE CONVERSATION SECTIONS

The $PROGRAM and $SUBROUTINE sections contain the statements which conduct the conversation with the user. There are four types of statements that are used. Text statements contain the exact text that is to be written out to the user at the terminal. Action statements contain commands which carry out functions of the program. Reaction statements are similar to action statements, but whether they are executed or not depends on the user's responses. Identifier statements are used for branching.

Sample Program and Execution

Here is an example of a partial PAL program for a simulation model of Michigan.

$MODEL MICMOD
$COMMON BLOCK
OUTSW
NPRT-THE NUMBER OF POINTS IN TIME TO BE PRINTED (6 PER RUN MAXIMUM)
TPRT(6)-TIMES TO BE PRINTED (ONE FOR EACH TIME TO BE PRINTED)
$PROGRAM MICIG
L WELCOME TO THE MICHIGAN PROJECTION MODEL. THIS MODEL IS BASED ON THE MSU PROJECT 8085 AND IS COMPOSED OF SIX COMPONENTS WHICH INTERACT TO PROJECT MICHIGAN AGRICULTURE TO 1985. POLICIES ARE IMPLEMENTED IN THE MODEL BY CHANGING PARAMETER VALUES, AND ARE GROUPED BY COMPONENT. FROM WHICH COMPONENT WOULD YOU LIKE TO SEE OUTPUT?
C 1. HUMAN POPULATION
   2. LIVESTOCK POPULATION
   3. LAND ALLOCATION
   4. PRODUCTION
   5. DEMAND
   6. ACCOUNTING
   7. NONE
1 >OUTSW=1
2 >OUTSW=2
3 >OUTSW=3
4 >OUTSW=4
5 >OUTSW=5
6 >OUTSW=6
7 >OUTSW=0
WOULD YOU LIKE TO SELECT PRINTOUT TIMES OTHER THAN THESE?  
1 >EXPLAIN (NPRT)  >READ (NPRT) 
>EXPLAIN (TPRT)  >READ (TPRT(1-NPRT)) 
2 >TPRT(1)=1970.  >TPRT(2)=1975.  >TPRT(3)=1980. 
>TPRT(4)=1985.  >NPRT=4 
L  OK, HERE WE GO.  
A  >RUN >EXIT 

If this conversation were to be translated and executed, this is what it would look like at the terminal. The user's responses have been underlined.

WELCOME TO THE MICHIGAN PROJECTION MODEL. THIS MODEL IS BASED ON THE MSU PROJECT 80&5 AND IS COMPOSED OF SIX COMPONENTS WHICH INTERACT TO PROJECT MICHIGAN AGRICULTURE TO 1985. POLICIES ARE IMPLEMENTED IN THE MODEL BY CHANGING PARAMETER VALUES, AND ARE GROUPED BY COMPONENT. FROM WHICH COMPONENT WOULD YOU LIKE TO SEE OUTPUT?
1. HUMAN POPULATION  
2. LIVESTOCK POPULATION  
3. LAND ALLOCATION  
4. PRODUCTION  
5. DEMAND  
6. ACCOUNTING  
7. NONE

*4  SIMULATION RESULTS ARE DISPLAYED FOR 1970, 1975, 1980, AND 1985. WOULD YOU LIKE TO SELECT PRINTOUT TIMES OTHER THAN THESE?
*YES  THE NUMBER OF POINTS IN TIME TO BE PRINTED (6 PER RUN MAXIMUM)
**3  TIMES TO 'E PRINTED (ONE FOR EACH TIME TO BE PRINTED)
**1977  **1982  **1987  
OK, HERE WE GO.

This would be followed by a table of output values produced by the model.

Sample Program Analysis

The sample program begins with a declaration of one of the common blocks that is in the model, BLOCK1. The variables OUTSW, NPRT, and TPRT comprise the block. The last two have explanations provided, which
are used later in the conversation. TPRT is an array which is dimensioned to 6.

After the $COMMON section is a $PROGRAM section. The first text statement to be printed is defined by the L in column 1. This indicates that the following text is the long text. This is the text that would normally be used. Other types of text will be discussed later. Note that the text ends with a multiple-choice type of question.

After the long text comes the choice text, indicated by the C in column 1. As you can see in the actual run of the program, the choice text is printed just as it appears in the program, directly following the long text. A single asterisk then appears at the terminal, telling the user to type in the number of his choice.

The statements that are numbered 1 through 7 in column 2 are called reaction statements. The number the user chooses determines the number of the reaction statement that will be executed.

A reaction statement may contain one or more commands. Notice that all commands begin with a greater-than sign (>). These particular reaction statements contain only assignment commands. Other commands are used later in the program. The assignment command does just what it would in FORTRAN; it assigns the value on the right hand side to the variable on the left hand side. The user in the example has picked choice 4, so OUTSW will be set to 4.

Following through the program, the next text statement simply requires a YES or NO answer, so no choice text is necessary. A YES answer will always correspond to reaction 1, a NO to reaction 2. The user chose YES, so reaction number 1 was executed. This time two different commands are involved. First the >EXPLAIN is executed, which prints the explanation of the variable. Then the >READ is executed. That is, the
user is asked to provide a value for this variable. The double asterisk 
appears and the user types in his value. Then the next >EXPLAIN and 
>READ are executed. This >READ specifies a range for the subscript. 
That means that values are to be read in for TPRT(1), TPRT(2), ..., 
TPRT(NPRT), one right after the other.

After that, the last line of text is written out. This text state­
ment is a little different than the previous ones, in that it does not 
ask a question. For this reason it does not need a set of reactions.

The last statement is an action statement. It consists of commands 
which are to be executed at this point in the program. It does not 
depend on a choice from the user; it will always be executed. It is 
denoted by an A in column 2. Here, two commands are to be executed. 
The >RUN calls and executes the model, and the >EXIT stops execution of 
the PAL program.

Text Statements and Modes of Conversation

As indicated above, the text statement can be used in three different 
ways. The first one is a multiple-choice question, which the user must answer 
with a number. The second is a yes or no question. The third type is 
declarative or descriptive, requiring no response from the user. The 
first two types must be followed by reactions, while the third type 
must not be.

After a user has executed the same conversation several times, he 
will no longer need to have all of the text written out. In fact, he 
will probably prefer a shorter version, in order to speed up the con­
versation. PAL allows the programmer to provide an abbreviated text.
Here is what the sample program might look like with short text.

$PROGRAM MICHIG
\[\text{L WELCOME TO THE MICHIGAN PROJECTION MODEL. WHICH WOULD YOU PREFER?}\]
\[\text{C 1. THE LONG CONVERSATION}\]
\[\text{2. THE SHORT CONVERSATION (NOT RECOMMENDED FOR BEGINNERS)}\]
\[\text{1 2} \text{CONVERSATION(S)}\]
L \[\text{THIS MODEL IS BASED ON THE MSU PROJECT 80&5 AND IS COMPOSED OF}\]
\[\text{SIX COMPONENTS WHICH INTERACT TO PROJECT MICHIGAN AGRICULTURE TO}\]
\[\text{1985. POLICIES ARE IMPLEMENTED IN THE MODEL BY CHANGING}\]
\[\text{PARAMETER VALUES, AND ARE GROUPED BY COMPONENT.}\]
B \[\text{FROM WHICH COMPONENT WOULD YOU LIKE TO SEE OUTPUT?}\]
C \[\text{1. HUMAN POPULATION}\]
\[\text{2. LIVESTOCK POPULATION}\]
\[\text{3. LAND ALLOCATION}\]
\[\text{4. PRODUCTION}\]
\[\text{5. DEMAND}\]
\[\text{6. ACCOUNTING}\]
\[\text{7. NONE}\]
\[\text{1 2 OUTSW=1}\]
\[\text{3 4 OUTSW=3}\]
\[\text{5 6 OUTSW=5}\]
\[\text{7 OUTSW=0}\]
\[\text{WOULD YOU LIKE TO SELECT PRINTOUT TIMES OTHER THAN THESE?}\]
S \[\text{DIFFERENT PRINT TIMES?}\]
C \[\text{1. YES}\]
\[\text{2. NO}\]
\[\text{1 EXPLAIN(NPRT) READ(NPRT)}\]
\[\text{2 TPRT(1)=1970. TPRT(2)=1975.}\]
\[\text{3 TPRT(3)=1980. TPRT(4)=1985.}\]
\[\text{NPRT=4}\]
L \[\text{OK, HERE WE GO.}\]
A \[\text{RUN EXIT}\]

The question above that asks from which component output is desired has a B in column 1. That indicates that the following text is both part of the long text and part of the short text. Thus the long text consists of the L text plus the B text, while the short text consists of only the B text.
The first question asks the user which text he wants. If the answer is the short text, the >CONVERSATION command is used to change the "mode" of the conversation from long to short. The conversation always starts out in the long mode, which prints the long text and choices. If there is no long text, the short text is printed. If the mode is changed to short, only the short text is printed. If the short text does not exist, then the long text is printed. So no matter what mode the conversation is in, something will always be printed for every text statement.

Occasionally you may want a particular text omitted entirely in the short mode. The only way to do this is to put an S in column one and leave the rest of the line blank. This will print a blank line at the terminal.

For more information, see the section "Text Statements and Modes of Conversation" in Appendix B of the PAL Reference Manual (CLASS-4).

There is one other type of text, the explanation text. If you feel a novice user may not find the long text sufficient, a further explanation may be provided. It is not automatically printed in either mode of conversation, and the user must ask for it by entering an E at the terminal. He does this when the computer is waiting for him to make a choice. Here is an example of an E field.

L WHICH TYPE OF FERTILIZER WOULD YOU LIKE TO APPLY?
E THE THREE COMPONENTS OF FERTILIZER ARE NITROGEN-PHOSPHORUS-POTASSIUM.
C 1. 6-24-24
2. 6-18-6
3. 4-10-10
Identifier Statements

Branching is accomplished in a PAL program by use of an identifier statement. This is comparable to a statement label in FORTRAN, but it must appear on a line all by itself. Here is an example.

L WOULD YOU LIKE TO EXPERIMENT WITH POTATOES?
C 1. YES
  2. NO
1
2 >GO TO CORN
L WHAT WOULD YOU LIKE THE PRICE OF POTATOES TO BE IN 1970?
A >READ(PRPOT(1))
L WHAT WOULD YOU LIKE THE PRICE OF POTATOES TO BE IN 1985?
A >READ(PRPOT(16))
CORN
L WOULD YOU LIKE TO EXPERIMENT WITH CORN?

Reaction Statements

Reaction statements are used wherever the user is to be given a choice between two or more options, and you want the program to execute different commands depending on his choice. The statements always appear in sets of 2 to 51, and they are always preceded by a text statement. The text statement must be there to ask him a question or explain the choices available to him.

Out of each set of reactions, only one statement will be executed each time. The statement that will be executed will be the one with the same number as the user's response. The commands in the statement are executed in the order they appear. When the last command in the statement has been executed, the other reaction statements in that set are ignored and control transfers to the statement following the set of reaction statements. A reaction statement may have no commands in it, in which case control will immediately transfer to the statement following the set of reaction statements.
Whenever a set of exactly two reaction statements is found, PAL will accept a YES or NO answer instead of a numeric one. In this case, an answer of YES or Y or 1 will result in execution of the first reaction statement. A NO or N or 2 will execute the second reaction.

See "Reaction Statements" in Appendix B of the PAL Reference Manual (CLASS-4) for more information.
CHAPTER IV
COMMAND USAGE

Some of the PAL commands require a little more explanation than is provided in the PAL Reference Manual (CLASS-4) in order to be used as effectively as possible. They are described here in alphabetical order. It is best to read the section in the Reference Manual about each command before reading the description here.

1. >CHECK

This command checks the current values of variables to see if their values meet the constraints specified. If no $CONSTRAINTS section appears in the PAL program, the >CHECK command will have no effect. This capability is provided primarily to catch ridiculous values which may be caused by typing errors or a misunderstanding of what kind of value is needed. Some values may even cause a machine error (such as dividing by zero).

The constraint statements cannot be checked individually. Every time a >CHECK is executed, every constraint will be checked. For this reason, a >CHECK will usually appear only once in a program. It will be at a point where all of the values that may be changed have been changed, and it will be before the >RUN command.

Normally the >CHECK will be the command immediately preceding the >RUN command. The exception to this will occur when type INITIAL variables are used. Then if the >CHECK was executed, the user would correct those illegal values and the model would run. But the values that had been stored would still be the illegal ones, which is not a desirable situation. So the sequence of commands should be:

A >CHECK >STORIV >RUN
See the >INITIALIZE and >STORIV commands for more information.

A constraint statement may not always be the best way to find illegal values. An unsophisticated user may have difficulty correcting his values when confronted with a constraint violation if he is unable to read FORTRAN logical expressions and use a few PAL commands. If there are particular places in your PAL program where a user is likely to enter illegal values, it may be better to write your own code to check the values as they are entered. This way you can do a much better job of explaining to the user what he did wrong.

2. >INITIALIZE and >STORIV

These commands are used only if a program contains type INITIAL variables. Otherwise, they have no effect. See "$TYPE Section" in Chapter II to determine if you need INITIAL variables.

If type INITIAL variables are declared in a PAL program, the translator sets up a table large enough to contain the values of all of these variables. This table will be used at the time the conversation is executed.

The >STORIV command takes the current values of all type INITIAL variables and stores them in the table mentioned above. The >INITIALIZE command takes the values from the table and reassigns them to the variables. Thus their current values will become the same as they were when the last >STORIV was done. Note that these commands do not deal with the standard value file at all. They deal only with current values (which, of course, may happen at times to be the same as the standard values).
A $\texttt{STORIV}$ command must be executed before the first time an $\texttt{INITIALIZE}$ is done. If not, the initial value table will not have any values in it. Then $\texttt{INITIALIZE}$ would cause the current values to be set to something unpredictable and probably illegal.

It is important to understand the difference between $\texttt{RESET}$ and $\texttt{INITIALIZE}$. The purpose of a $\texttt{RESET}$ is to undo all changes the user may have made to program variables. It does this by replacing all current values with standard values. This means the user is starting all over as far as setting values is concerned. The purpose of an $\texttt{INITIALIZE}$ is to undo all changes to variables which may have been caused by running the model. This means the user is starting again with the values which were current just before the model was run. These may or may not be the same as the standard values. He may then make further changes and run the model again. In contrast, if a $\texttt{RESET}$ were done, he would lose all of the nonstandard values he had used for the previous run.

3. $\texttt{PRINT}$

The $\texttt{PRINT}$ statement in PAL is quite easy to use because it does not require anything like a FORTRAN FORMAT statement. However, if you want specific spacing to print labels or headers, it can be complicated. The techniques described here may simplify things somewhat.

A new print line is started whenever a new $\texttt{PRINT}$ command is found or when the previous print line is full. This means that one $\texttt{PRINT}$ command, with five variables specified in its argument list, will print one line of values, while five consecutive $\texttt{PRINT}$ commands, each containing one variable, will print five separate lines.
Each numeric value will be right-justified in one of the six 12-column fields predefined by PAL. Each HOLLERITH value will be printed with an (A10) format and will start in the column immediately following the previous value. Suppose, for example, you used one >PRINT command to print one numeric value, three HOLLERITH values, and another numeric value. The first numeric value would be right-justified in the first field, ending in column 12. The three HOLLERITH values would occupy the next 30 columns, column 13 through column 42. The last numeric value would be right-justified in the next available field, column 49 through column 60.

Suppose you wanted headings for each of the six numeric fields. One way to do this would be to write a text statement before the >PRINT command. This text statement would be the entire header line and could be spaced accordingly. Another way would be to print a line of HOLLERITH variables. You could not, however, print six HOLLERITH variables and expect them to line up with the six numeric fields because HOLLERITH variables are printed with no blanks in between. Each one begins where the previous one ended. What you must do then, is to print a full line of HOLLERITH variables, including blanks exactly where you want them. Probably the best way to do this is to declare a LOCAL HOLLERITH array and initialize it to the desired HOLLERITH values.

It is important to remember that local variables do not have standard values. A >PRINT command will always print the current value of a local variable, even if the S argument is used.

Another thing that cannot be done directly in PAL is to print standard and current values on the same output line. This is because a single >PRINT command prints all current or all standard values, and each new >PRINT command starts a new line. The way to get around this
is to use a dummy local variable as follows:

```
A >DUM = LENGTH >PRINT(S,LENGTH,DUM)
```

The current value of DUM will be equal to the current value of LENGTH and, because it is a local variable, the >PRINT will automatically print its current value. So the first value on the line will be the standard value of LENGTH and the second value will be the current value of LENGTH.

4. >READ

When using >READ, it is important to remember that HOLLERITH variables are handled differently than other variables. Normally only one value is read in from each line. However, if a range of values in a HOLLERITH array is to be read, it is assumed that they are intended to be used as a label of some sort, i.e., that they form a single phrase. In this case, all of the values in the range will be read from one line on the terminal.

Each input line must be no longer than 72 characters and an (A10) format is used for each value, so each HOLLERITH array range should represent no more than 7 values. If a larger range is used, an error message will be printed at execution time. The PAL translator does not check for this.

To avoid confusion and mistakes on the part of the user, the conversation should make it quite clear what type of value is expected for each >READ. If a HOLLERITH value is requested, the user needs to know how many characters long it should be. Often a good way to do this is to use an L text field to explain this. It is usually safe to assume that anyone who uses the short mode of conversation will not need this information and will probably not want it.

The >READ command behaves slightly differently than the >PRINT, in that it makes no difference whether you put a whole list of variables in
one >READ or have a series of >READ's, each containing one variable. The
result that the user sees will be the same in either case.

5. >REQUEST

This command is unique in that it can be used only in an action
statement, not in a reaction. The reason for this is that >REQUEST is
designed to replace a set of reactions of a particular form. Suppose a
PAL program contained the following statements:

L WHAT IS YOUR ANNUAL FAMILY INCOME?
C 1 $0-4000
  2 $4001-8000
  3 $8001-12000
  4 $12001-20000
  5 OVER $20000
1 >INCGRP=1
2 >INCGRP=2
3 >INCGRP=3
4 >INCGRP=4
5 >INCGRP=5

This is the type of situation that >REQUEST was designed for. The
set of five reactions can be replaced by a single action as follows:

A >REQUEST(INCGRP,5)

Notice that the reaction statements must be removed entirely. The
>REQUEST actually takes the place of the reaction statements. It is not
correct to simply leave the reaction statements blank and follow them
with the action statement containing the >REQUEST. In other words, if
a set of reactions is used in conjunction with a particular text state-
ment, a >REQUEST must not be used; if a >REQUEST is used with a particular
text, reaction statements must not be used.

These two different ways of writing this code will do exactly the
same thing and will look the same to the user at the terminal. The only
difference is that the >REQUEST is easier for the programmer to write.
The larger the number of reactions needed, the more you save by using >REQUEST.

Notice that the purpose of the >REQUEST is very specific. It works best in the situation above, where all of the reactions do exactly the same thing and the value assigned to the variable is the same as the reaction number. The further a set of reactions is from fitting these criteria, the less useful >REQUEST is for replacing the reaction L.

Here are two more examples of uses of >REQUEST:

L WHAT COMMODITY WOULD YOU LIKE TO MAKE CHANGES TO?
C 1 CORN 2 WHEAT 3 BARLEY
   4 SOYBEANS 5 RICE 6 NONE
1 >INDCRP=1
2 >INDCRP=2
3 >INDCRP=3
4 >INDCRP=4
5 >INDCRP=5
6 >GO TO LAST

These reactions can be replaced with:

A >REQUEST(INDCRP,6) >IF(INDCRP.EQ.6)GO TO LAST

Sometimes the values desired may not be the same as the reaction numbers.

L WHAT TYPE OF TAX WOULD YOU LIKE TO IMPOSE FIRST?
C 1 NO TAXES
   2 SALES TAX
   3 PERSONAL INCOME TAX
   4 BUSINESS INCOME TAX
   5 VALUE-ADDED TAX
   6 IMPORT TAX
1 >ITAX=0 >GO TO NTAX
2 >ITAX=1 >CALL TAX
3 >ITAX=2 >CALL TAX
4 >ITAX=3 >CALL TAX
5 >ITAX=4 >CALL TAX
6 >ITAX=5 >CALL TAX

These reactions could be replaced with:

A >REQUEST(ITAX,6) >ITAX=ITAX-1 >IF(ITAX.EQ.0) GO TO NTAX >CALL TAX

25
For an example of a case where \texttt{REQUEST} is not useful, suppose the reactions above were changed to:

\begin{verbatim}
1 >ITAX=0 >GO TO NTAX
2 >ITAX=1 >CALL SALTAX
3 >ITAX=2 >CALL INCTAX
4 >ITAX=3 >CALL BUSTAX
5 >ITAX=4 >CALL VALTAX
6 >ITAX=5 >CALL IMPTAX
\end{verbatim}

Here the reactions cannot be replaced because each of them does a different thing.

This has shown how \texttt{REQUEST} can be used to replace a set of reactions. There are also some instances where it can be used instead of a \texttt{READ}. In fact, it may be preferable to a \texttt{READ} because it checks the user's value when he enters it and, if it is illegal, asks him for a new one. Of course, it can only be used for INTEGER or REAL variables which are to have integer values from 1 up to some specified maximum. If, for example, you wish to allow a zero value or a fractional value, \texttt{REQUEST} cannot be used. Suppose you had the following statements:

\begin{verbatim}
L FOR HOW MANY DIFFERENT YEARS (UP TO 6) DO YOU WANT RESULTS PRINTED?
 A >READ(NPRT)
L ENTER THE YEARS (ONE PER LINE) FOR WHICH YOU WANT RESULTS PRINTED.
 A >READ(TPRT(I-NPRT))
\end{verbatim}

If TPRT is dimensioned to 6 and the user mistakenly enters a value for NPRT larger than 6, then the next \texttt{READ} will store something outside the bounds of TPRT. If the first action were:

\begin{verbatim}
 A >REQUEST(NPRT,6)
\end{verbatim}

the user will not be allowed to enter a value less than 1 or greater than 6. This makes the \texttt{REQUEST} much more desirable than the \texttt{READ} in this case.
6. >RUN

The >RUN command, when used from the terminal, is different than all other PAL commands. The >RUN command may be used only when the PAL program is operating in command mode. All other commands may be entered any time the user is asked for a response, even in conversation mode. The reasons for this are explained in Chapter IX.

There are also restrictions on where a >RUN command can appear in a PAL program, if that program is overlayed. These restrictions and the reasons for them are explained in Chapter IX.
CHAPTER V

THE STANDARD VALUE FILE

Every PAL program which contains one or more $COMMON sections, must have a corresponding standard value file. The standard value file contains default values for all of the common variables.

The standard value file is used in three ways. It is read in immediately prior to the start of the conversation so that all common variables are initialized to their standard values. It is read in whenever a >RESET command is executed. It is also used for printing standard values.

Note that local variables have only current values, not standard values. The current values may be initialized by using the local variable statement, but values for them must not be included on the standard value file. For this reason, a >RESET command has no effect on local variables.

The standard value file contains the standard values for each variable in each $COMMON section in the order in which they are declared in the PAL program. The file is binary (created by an unformatted FORTRAN write statement), with 63 central memory words per logical record. If the size of a common block is not exactly divisible by 63, the last variable of the common block will be written in the middle of a logical record. Rather than starting the next common block in the middle of a logical record, the remainder of that record is not used. Because short records are not allowed, the remainder of that record must be filled with something (anything). The next common block will start at the beginning of the next logical record.

Creating a Standard Value File

There are several ways in which a standard value file can be created. A separate FORTRAN program may be written to create the file. For each
$COMMON section in the PAL program, the FORTRAN program should contain
an array dimensioned to the length of that section. The values to go
on the file should be put into the array(s), perhaps in DATA statements
or by reading data cards. The values within each array must be in the
same order as the variables in the PAL $COMMON section.

Once the correct values are in the array(s), a FORTRAN subroutine
called SZZCRE can be called. SZZCRE is part of the PAL package and is
available on a permanent file. There must be one call for each $COMMON
section. SZZCRE will take the array(s) and write out a standard value
file in the proper format. Three arguments must be provided in the call
to SZZCRE. The form of the call should be:

CALL SZZCRE(ARRAY,NSIZE,LUNIT)

where ARRAY is the array of data values, NSIZE is the dimension of ARRAY,
and LUNIT is the number of the logical unit you would like the standard
value file to be written on. LUNIT will be the same for all of the calls.
The order of the calls must correspond to the order of the $COMMON sections
in the PAL program.

ARRAY may be real or integer, but the values stored in it must have
the proper internal representation for the type of the variables they
correspond to. For example, suppose you decide to use a real array
called DUMMY, and suppose the first variable on your file is an integer
variable, NPRT, with a desired value of 4. Suppose you put its value
into DUMMY with the following statements:

DUMMY(1) = 4

The value would be converted to real representation before it was
stored. It would be written this way on the standard value file by
SZZCRE. Then, when it was read back in to run the conversation, it would
be treated as an integer value. This could cause many errors in your program. There are several ways to avoid this. **EQUIVALENCE** statements can be used with variable names of the correct type. **DATA** statements and **READ** statements will generally allow you to put a value of one type in a variable of a different type.

If you have a working model, there is an easier way to create the file. The model will already be using the proper common blocks and, at some point near the beginning of a run, the variables should all have their standard values. At that point, you should insert calls to SZZCRE to create the standard value file. There should be one call for each $COMMON section. The first argument in each call should be the first variable in the corresponding common block. As before, you must include the size of the common block and the logical unit number. The order of the calls must correspond to the order of the $COMMON sections in the PAL program. The calls should be followed by a **STOP** statement, unless a full run of the model is desired at the same time. After the model finishes, you can catalog the newly created standard value file for use with the conversation. Here is an example of a model set up to write a standard value file:

```
PROGRAM MICH(TAPE1,TAPE7)
COMMON /BLOCK1/ TPRT,NPRT,DAIRY(17), GRAINS(17), FRUIT(17)
COMMON /BLOCK2/ DEMAND, PROD,LNDGRN(12)
DATA NPRT,TPRT /4,1970./
READ(1) DAIRY,GRAINS,FRUIT
READ(1) DEMAND,PROD,LNDGRN
CALL SZZCRE(TPRT,53,7)
CALL SZZCRE(DEMAND,14,7)
STOP
.
.
END
```
When using either method of creating the standard value file, you should make sure that all variables have their desired values at the point where SZZCRE is called. Failure to do this may result in constraint violations or errors at execution time.

Listing the Standard Value File

After the standard value file has been created, there may be times when you will want to know what the values on it are. You may wish a list of values to see if it was created correctly or just because you forgot what some value was. There is a program called STDWRT that will list your standard value file for you. It is part of the PAL package and is available on a permanent file.

In order to run STDWRT, you will also need the first three FORTRAN subroutines that PAL produces when it translates your PAL program. These contain information, such as variable names and dimensions, that is needed to print out the values. The names of the three routines are: SZZPAL, SZZDAT, and SZZINC. They will always be the first three subroutines of a compiled conversation, so you can copy them to another file easily. See Appendix B for a full description of the FORTRAN routines that PAL produces.

The standard value file to be used as input should be on unit 7. Unit 9 will contain the output, which will be a listing of variable names and their values. You can copy it to a printer if you want it printed, or you can save it for later use. There will be a blank at the beginning of each line for carriage control.

Each variable to be printed will start on a new line. For arrays, six values will be printed per line. Each line will be 72 columns wide.
Each numeric value will be right-justified in a 12-column field. Each HOLLERITH value will be printed with a (2X,A10) format. Arrays will be printed with the first subscript varying fastest.

Changing Values on the Standard Value File

There may be times when you want to change some of the values on the standard value file. You might, for example, come up with a better estimate for some parameter in the model and want to replace the old value on the standard value file. You might want to make several copies of the standard value file with slightly different values representing different base runs. Or you might be using a conversation at a terminal and wish to save the changes you have made to the current values to be used for a run at some other time.

The >SAVE command will allow you to do these things easily. You begin by executing the PAL conversation, using the old standard value file. You change the values that you want changed, either by entering commands or by following through the conversation. When all the variables have the desired values, you enter the >SAVE command. This takes all the current values and writes them on logical unit 8 in the correct form for a standard value file. You can then continue using the conversation and running the model, or you can enter an >EXIT command to end the conversation. When you are finished you must catalog TAPE8 for later use.

You may use the >SAVE command as many times as you wish during a single execution of the conversation. However, the saved file is always rewound before it is written on. This means that at the end of execution only the last set of saved values will be on the file. If you want to
save more than one set of new values, each must be saved in a separate run of the conversation.

It is best to enter a >CHECK command just before you use the >SAVE. This will insure that illegal values are not written on the file.

There is another method of changing values on the standard value file. This method is usually less convenient than use of the >SAVE command. Its most useful application is for changing the size of a standard value file. How to change the size of the file will be described in the next section.

This method of changing values uses two separate FORTRAN programs. One is STDWRT which was described in the previous section. The other is called STDCHG, and it uses the output from STDWRT as its input file. Then it writes out a new standard value file.

You must first run STDWRT as described above and save the coded output file with the names and values. You can then make changes to that file, putting in the new values that you want. STDWRT writes the variables in a particular format, and you must be careful to maintain that format so that the file will be read back in correctly. The line with the variable name, and the blank lines before and after it, should not be changed. No lines should be added. The formats for reading the values back in will be (2X,A10) for HOLLERITH variables, (I12) for INTEGER variables, and (E12.0) for REAL variables.

STDCHG also requires the first three subroutines from the compiled PAL conversation. It reads the corrected coded file from unit 8. From this it produces a new standard value file on unit 7. It then reads this file back in and writes out a new coded version of the file on unit 9.
This is done just so you can check the values to make sure they are what you intended them to be. This file is in the same format as that produced by STDWRT.

Here is a summary of how you would change the standard value file:

**Step I:** Run STDWRT. The files it uses are:
- logical unit 7--(input) the standard value file in binary form (without corrections)
- logical unit 9--(output) the standard value file in coded form (without corrections)

**Step II:** Make corrections to the coded file (unit 9). The new file would become logical unit 8 in Step III.

**Step III:** Run STDCHG. The files it uses are:
- logical unit 8--(input) the standard value file in coded form (with corrections) from Step II
- logical unit 7--(output) the standard value file in binary form (with corrections)
- logical unit 9--(output) the standard value file in coded form (with corrections)

This technique will work only when you are changing the values that are already on the standard value file. It will not allow you to add or delete variables or rearrange the order of the variables. To do this, see the next section.

---

**Changing the Size of the Standard Value File**

Adding or removing values on the standard value file can be done using STDWRT and STDCHG, but it is a little more involved.

Using your old standard value file as input, run STDWRT to create the coded file. This will require the first three subroutines from your old PAL program.
Then take the coded file of values and make the desired changes to it. Remember that the format of the file must be maintained. For each variable there must be two blank lines, a line with the variable name, another blank line, and then the values, six per line in the appropriate format. So for each variable you add or remove, there will be at least five lines involved (more for some arrays). If you do not get this exactly right, the program probably will not work.

Next you must take your PAL program and change the $COMMON sections to reflect the new number of variables. Then you must recompile it because you will need the first three subroutines from it for the next step.

Now you are ready to run STDCHG with the corrected coded file as input. Use the first three subroutines from the new PAL program. STDCHG will produce a new standard value file on unit 7. It will also write a list of the new file on unit 9. You should always check this list to make sure that it matches your input file (unit 8). This will insure that all the values were read in correctly.
CHAPTER VI
DEBUGGING A PAL PROGRAM

Errors may show up in a PAL program in any of the three following ways:

1. They may be detected by the PAL translator.
2. They may be detected by the FORTRAN compiler.
3. They may be detected at execution time.

PAL Translation Errors

During translation of a PAL program, the translator writes on two files. The PAL source code is listed on the standard output file. Messages intended to be displayed at a terminal are written on a message file. See Appendix A for further information about these files.

Any error messages are interspersed with the source code listing on the output file. A message will always refer to the last statement (not including comments) printed before the message. The only exception to this is messages which list unreferenced and undefined identifiers. These messages will be listed at the beginning of the $PROGRAM or $SUBROUTINE following the one to which the messages refer.

Fatal error messages are preceded by a line which looks like this:

>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>FATAL ERROR<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<

Any fatal error will result in no FORTRAN code being produced. Nonfatal errors are simply printed where they occur, with no preceding line. If only nonfatal errors are found, the object code will still be produced and will run properly. The nonfatal error messages are used to point
out things which the programmer may not have done the way he intended.

Separate counts are kept of the number of fatal and nonfatal errors. These are printed at the end of the output file, after the variable table. They are also printed on the message file so that you can tell from a terminal whether there were errors, without listing the entire output file. If there is no message on the message file about errors, then none were found.

There are three very common errors which cause many PAL syntax errors.

The first error is putting something in the wrong column. There are very specific rules about what must appear in the first four columns. Here is a short review of those rules. A continuation card must be blank in columns 1-4. A text statement must have an L, S, B, E, or C in column 1 with blanks in 2-4. An action statement must have an A in column two, with blanks in 1 and 3-4. A reaction statement must have a number from 1 to 51 starting in column 2, with columns 1 and 4 blank. Column 3 must be blank unless the number has two digits. An identifier must be 2 to 4 characters long, starting in column 1, with the rest of the card blank. All statements not mentioned above must start in column 1.

The second error involves the illegal use of blanks. A few statements require blanks in particular places; other statements do not allow them. If the PAL translator indicates a statement is illegal, and the problem is not immediately obvious, check for extra or missing blanks.

The third error is misuse of the greater-than sign (>) for commands. Every command must begin with a greater-than. The exception to this rule is any command which follows the logical expression of an >IF
command. This must not begin with a greater-than.

It is very important to remember that when PAL finds an error in a statement, it ignores the remainder of the statement. This means that even if you fix all of the errors that the translator finds on one run, on the next run there may be new errors in parts of the code that the translator ignored the first time.

This failure to process a statement beyond the first error will occasionally result in false error messages. For example, if a >GO TO command is not processed because of an error earlier in the same statement, you may get a message which says that the identifier in the >GO TO was unreferenced. Another example of this occurs when there is a set of two reaction statements with an error in the first one. Then the translator does not recognize the first reaction and it will say that there must be at least two reactions.

Sometimes when the translator encounters an error, it may not know exactly what is wrong. It will print an error message, but the message may not make it obvious what is wrong. Some of the more common causes of this are missing commas or parentheses, missing or extra blanks, and variable names or expressions that are too long.

FORTRAN Compilation Errors

There are portions of certain PAL statements which the PAL translator simply copies into the FORTRAN object code, without checking for syntax errors. Virtually every FORTRAN compilation error that you find will be a result of one of these statements. The main problem when dealing with a FORTRAN error is that of determining which PAL statement it corresponds to.
First, you should check to see if the code that caused the error was inserted using a $FORTRAN directive. The PAL translator does not check any of this code, and it is very easy to get in trouble this way.

If this was not the problem, then determine what type of FORTRAN statement caused the error and what routine it is in. Then Table 1 (see next page) will tell you what type of PAL statement you should look for. This should help you find the error.

It is also possible to get a FORTRAN diagnostic indicating that a certain statement cannot be executed because control will never be transferred to it. This means that there is a corresponding PAL statement which cannot be executed. The way to identify this is to look at the FORTRAN statements and use the table to try to identify the PAL statements which produced them.

Execution Errors

Errors in execution are usually more difficult to diagnose than errors in compilation. There are too many different symptoms to list them all here, and, in fact, the symptoms of a particular problem may vary from one machine to another. For this reason, some of the possible causes of execution errors, not the symptoms, are listed. This should help you pinpoint problems to a certain extent.

Some possible causes of execution errors are:

1. Standard value file errors, such as incorrect number of values on the file, incorrect values, or incorrect positioning of values (relative to the $COMMON section ordering of variables).

2. Improper use of $FORTRAN directive.

3. Incorrect name on $MODEL card.
<table>
<thead>
<tr>
<th>FORTRAN statement</th>
<th>FORTRAN routine</th>
<th>PAL statement</th>
<th>PAL section</th>
</tr>
</thead>
<tbody>
<tr>
<td>DATA</td>
<td>SZZPAl</td>
<td>local variable statement</td>
<td>$LOCAL</td>
</tr>
<tr>
<td>IF</td>
<td>SZZCNS</td>
<td>constraint statement &gt;IF</td>
<td>$CONSTRAINTS **</td>
</tr>
<tr>
<td>CALL SZZINR (...)*</td>
<td>any routine</td>
<td>&gt;READ</td>
<td>**</td>
</tr>
<tr>
<td>CALL SZZINC (...)*</td>
<td>any routine</td>
<td>&gt;PRINT</td>
<td>**</td>
</tr>
<tr>
<td>CALL</td>
<td>SZZRUN</td>
<td>Model name &gt;CALL</td>
<td>$MODEL **</td>
</tr>
<tr>
<td>variable=LZZANS</td>
<td>any routine</td>
<td>&gt;REQUEST</td>
<td>**</td>
</tr>
<tr>
<td>variable=FLOAT (LZZANS)</td>
<td>any routine</td>
<td>&gt;REQUEST</td>
<td>**</td>
</tr>
<tr>
<td>variable=expression</td>
<td>any routine</td>
<td>assignment command</td>
<td>**</td>
</tr>
</tbody>
</table>

* There will be one CALL for each variable used in a >PRINT or >READ command, so each command will often result in more than one CALL.

** The error in this case will be in the PAL $PROGRAM or $SUBROUTINE with the same name as the FORTRAN routine.

TABLE 1
4. Incorrect subroutine name in >CALL command.

5. A PAL $COMMON section which does not match the corresponding FORTRAN common block.

6. Illegal subscript value.

7. Using illegal variable name or function name in a subscript, a constraint statement, an >IF command, a >CALL command, or the right-hand side of an assignment command. PAL does not check the legality of variable names in these places.

8. Failure to do a >RESET, >STORIV, or >INITIALIZE at the proper time.
CHAPTER VII
PREPARING A MODEL FOR USE WITH A PAL CONVERSATION

There is usually a simulation model associated with a PAL conversation. One purpose of the conversation is to explain this model to the user. It does not have to be a simulation model. Any FORTRAN program may be used, but here we will always refer to it as the model. Usually the model will have to be modified somewhat in order to be run with the PAL program. This chapter explains what may need to be modified. If the model has not yet been written, this should help you in writing it.

Generally there is a simulation time range associated with a model. For example, it might cover the years 1960-1980, or perhaps an unspecified year consisting of 52 weeks, or four seasons a year for five years. There are two different ways to run a model of this type. The model may cover its entire time span each time it is called and then return to the PAL program. The user would then be offered the choice of another run, which would mean the model would be run again from the beginning. The other method of running the model would cover only a portion of the time span at a time. It would return to the PAL program which would give the user a chance to change some variable values. Then the model would be run again, starting where it left off and covering the next time span. It could be stopped in the middle any number of times. This method is more complicated, and it is mentioned here just to point out that the model and the PAL program must be made compatible in this respect.

The model will not be running by itself if a PAL program is used. It will be called from some point in the PAL program. This means that
the main program of the model must be made into a subroutine. The name of this subroutine is the name that must appear on the $MODEL card. At the point where the execution of the model would normally end (usually a STOP statement), you should use a FORTRAN RETURN statement. This will return control to the PAL program after the model is run.

When the model is run with the PAL program, all of the $COMMON variables will have their values supplied on the standard value file. Any statements in the model (such as READ, DATA, or assignment statements) that give initial values to these variables should be removed. If they are not removed, they may override any initial values the user has chosen for those variables in the conversation.

The model should then consist of a set of subroutines which are executed together to produce the model output. A model which consists of programs that run separately from each other, perhaps communicating via data files and requiring specific actions between execution of the separate programs, is not well suited for use with PAL. If you want to use the programs as a single model, you should combine them to form a single set of FORTRAN subroutines. Otherwise, each separate program should be considered to be a model, and one PAL program will be required for each one.

You must also create a standard value file for use by the conversation. The chapter on the standard value file explains how to do this.

You should identify all variables which must be declared type INITIAL. Failure to do this can produce errors in the results. See the explanation of type INITIAL in the discussion of the $TYPE section in Chapter II.
The PAL conversation uses standard input and output unit numbers which the model should conform to. Unit number 7 is used for the standard value file, so it should not be used at all by the model. Unit number 5 is used for input and should probably not be used by the model. Any input done by the model should be done from another file or changed so it can be done through the conversation. Unit number 6 is used as the output file. Any writing done by the model which you want to appear at the terminal should be done on unit 6.
CHAPTER VIII
INSERTING FORTRAN IN A PAL PROGRAM

For most conversation programs, the flexibility allowed by PAL will be sufficient. However, in certain instances, you may want a program to do something that is awkward or impossible in PAL. Under these circumstances, you may use the $FORTRAN directive to insert FORTRAN statements in your conversation program.

The PAL translator produces a FORTRAN program which is equivalent to your PAL program. When you use a $FORTRAN directive, the FORTRAN statements which follow it are copied directly into the FORTRAN program which the translator is in the process of creating. Your FORTRAN statements are not checked for errors. Chapter 8 of the PAL Reference Manual (CLASS-4) explains where the $FORTRAN directive may and may not be used.

Statement Numbering

One problem you will encounter when inserting FORTRAN code is that of statement numbering. You must not repeat any statement numbers that are used in the PAL-generated code. This can be done by following the instructions below.

All of the statement numbers used when translating the $CONSTRAINTS section will have a last digit of 0, 1, 2, or 3. This means that in FORTRAN insertions in the $CONSTRAINTS section you can use any statement numbers ending with 4 through 9.

In a $PROGRAM or $SUBROUTINE section, you can always use any statement
number whose last two digits are 52 through 58. If this is not satisfactory for some reason, you can determine the lowest allowable value for any particular section. Find the longest set of reactions in that routine. Then you can use any statement number whose last two digits are larger than the number of reactions in that longest set and less than 58. If the longest set had five reactions, any statement number ending in 06 or more could be used. If the longest set had twenty reactions, any statement number ending in 21 or more could be used. However, with this method, if you add a new, longer set of reactions later, you may create a conflict with statement numbers you have previously added. Another riskier rule of thumb is that unless the section is extremely long, any number containing five digits will work. If you want to be as safe as possible, use only numbers ending in 52 through 58.

Special Purpose Variables

These are a few variables in the FORTRAN conversation which are used for special purposes. These may be useful to you if you are inserting FORTRAN code of your own. It is not recommended that you change the values of these variables at any time. Following are explanations of them.

LZZOUT--This is the logical unit number of the standard OUTPUT file. It can be used if you want to print output at the terminal.

LZZANS--This contains an integer number which is the user's answer to either the most recent set of reactions or the most recent >REQUEST command. If he were given five choices and picked number three, LZZANS would be 3. If he answers a question YES, LZZANS will be 1; if he answers NO, it will be 2.

LZZMOD--This tells what conversation mode is currently in use. A 1 indicates LONG mode, a 2 indicates SHORT mode, and a 3 indicates COMMAND mode.
These variables can also be used in certain PAL statements. Variable names in most PAL statements are checked to make sure they are declared as common or local variables in your program. However, some statements are not checked for illegal names. This means that these variables can be used in the following contexts:

1. Constraint statements
2. Subscripts
3. Right-hand side of assignment command
4. Logical expression in a >IF command

Specific Uses of $FORTRAN

There are many possible ways of using the $FORTRAN directive. Four possibly useful ways are described below.

Every time a >CHECK command is executed, all of the constraints are checked. There are times, however, when a certain constraint may not apply. In that case, the statement can be skipped over, using the $FORTRAN directive. For example:

```
$FORTRAN
   IF(NPRT.LT.2) GO TO 25
$PAL
FOR I=2,NPRT REQUIRE TPRT(I-1).LT.TPRT(I)
$FORTRAN
25 CONTINUE
$PAL
```

In this case, if NPRT is less than 2, the constraint on TPRT is ignored.

The PAL translator normally declares four files on the PROGRAM card it generates. These are: TAPE5 for input, TAPE6 for output, TAPE7 for standard values, and TAPE8 for saving current values. It is possible that your model may require additional files for some purpose. If so, you
can insert your own PROGRAM card. To do so, there must be no symbolic name on your $PROGRAM directive. This tells the translator not to generate a FORTRAN PROGRAM card.

You must always include on your PROGRAM card certain files that are used by the library routines. It is recommended that you set the buffer sizes to 129. In this case, your file list would start as follows:

```
TAPE7=129, INPUT=129, OUTPUT=129, TAPE8=129, TAPE5=INPUT, TAPE6=OUTPUT
```

If you do not use this exact sequence of file declarations, you may run into trouble. You should follow this list with your own file declarations. If the model uses units 1 and 2, the sequence of cards would look as follows:

```
$END
$FORTRAN
PROGRAM X(TAPE7=129, INPUT=129, OUTPUT=129, TAPE8=129, TAPE5=INPUT, TAPE6=OUTPUT, TAPE1, TAPE2)
$PROGRAM
```

The PAL >PRINT command is somewhat limited in flexibility. Another use of the $FORTRAN directive is to insert WRITE and FORMAT statements to take advantage of FORTRAN's flexibility. One limitation to this is that only current values can be printed this way. Standard values can be printed only with the >PRINT command. LZCOUT should be used as the output unit number for any WRITE statement.

The $PROGRAM directive is also useful when overlaying a PAL program. The proper method is described in the next chapter.
CHAPTER IX

OVERLAYING A PAL PROGRAM

Many simulation models that are likely to be used with PAL require so much core that they must be overlayed. Even when this is not the case, the core required for a PAL program and library routines, when added to the size of a simulation model, may make overlaying a necessity. This chapter deals with overlaying PAL programs on a CDC 6500 computer. Many of the details will not be applicable to other computers. It is assumed that the reader already knows how to overlay a FORTRAN program and has read Chapter VIII and Appendix B of this manual.

Overlay Structure

There are many different ways a PAL program can be overlayed. The only requirement is that the first PAL $PROGRAM section must always be in the main overlay. This is because this section is always the first part of the conversation to be executed, i.e., it is the overall executive program for the run. It is convenient to classify overlayed programs into two types, according to the overlay structure.

The simplest type has the entire PAL program in the main overlay. There will also be at least one subroutine from the model in the main overlay. A run of the model will simply require a call to this "executive subroutine" which will call in various parts of the model as overlays. The entire conversation and PAL library routines will remain in core at all times. This, of course, works best when the conversation is not too lengthy.

The second type of overlay structure has a PAL program which consists of more than one overlay. This means that the PAL executive $PROGRAM will
call in various overlays to conduct portions of the conversation. Some portions may not even be used for a particular session because the user may not be interested in certain aspects of the model. A run of the model will then call in another overlay, replacing part of the conversation. A great deal of space may be saved this way, but the programming becomes slightly more complicated.

**Overlay Calls**

If the entire PAL program is in the main overlay, there will be no need for any overlay calls within it. A run of the model will result in a call to the subroutine specified on the $MODEL card, which will contain all of the necessary overlay calls.

If the PAL program consists of more than one overlay, each portion of the conversation must be loaded and executed with a command of the form:

>CALL OVERLAY(...)  
This would contain appropriate arguments for each conversation overlay.

If the model is a separate overlay, the $MODEL card should be of the form:

$MODEL OVERLAY(...)  
This would contain appropriate arguments for the call to the model. Then any >RUN command will result in a call to that overlay.

**Overlay Directives**

The programmer must also insert the loader OVERLAY cards in front of the appropriate binary decks. This can be done in three ways. First, the programmer can deal only with the binary decks. He can go through all
of the compilation stages for the model and the PAL conversation, and then insert OVERLAY cards where needed in the binary decks. The second method would be to insert the OVERLAY cards into the FORTRAN decks. The PAL program would be compiled into FORTRAN before this was done.

The third method, which is the recommended one, is to insert all OVERLAY cards that are needed in the conversation with the use of the $FORTRAN directive. The OVERLAY cards for the model would be inserted in the FORTRAN program of the model. Thus all OVERLAY cards will be in the original source decks, whether PAL or FORTRAN.

The first PAL $PROGRAM, the $CONSTRAINTS section, and the utility routines generated by the PAL translator must always be in the main (0,0) overlay. This means that the OVERLAY card for the main overlay must always be inserted before the $CONSTRAINTS section or, if this is not present, before the first $PROGRAM or $SUBROUTINE. If it is inserted before the $CONSTRAINTS, there must not be another OVERLAY card until after the first $PROGRAM (because they must both be in the main overlay). The sequence of cards must be

$FORTRAN
OVERLAY(...,0,0)
$CONSTRAINTS or $PROGRAM or $SUBROUTINE

When inserting the (0,0) overlay card, the $FORTRAN directive must not be preceded by an $END directive. This is explained below.

The remaining OVERLAY cards (not needed if the conversation is all one overlay) should be inserted before the appropriate sections of the PAL program in a similar way. The difference is that the $END directive must be used here. The sequence of cards would be:

$END
$FORTRAN
OVERLAY(...) 
$PROGRAM or $SUBROUTINE
Although the use of the $END directive as outlined above may not make sense, there is a logical reason for it. The $END directive, when found in a $PROGRAM or $SUBROUTINE section, tells the translator that the section is finished. The translator then finishes up the odds and ends involved in generating FORTRAN code, including writing a FORTRAN END statement. The same thing happens if a new $PROGRAM or $SUBROUTINE directive is found. However if a $FORTRAN directive is found, it is not assumed to be the end of the previous section. The FORTRAN code is simply copied into the generated code because it is assumed to be part of the section in which it is found. So if you try to insert an overlay card (other than the first one) without using an $END card, the overlay card will wind up in the middle of a FORTRAN routine.

Now for an explanation of why the first overlay card is different. The (0,0) card must be the first record on the FORTRAN file that is produced. There is only one way to make sure that it gets written there. The first $PROGRAM, $SUBROUTINE, $CONSTRAINTS, or $END directive indicates the end of the nonexecutable PAL code. When this is found, the translator immediately writes out several utility FORTRAN routines, which are needed for execution of the conversation. If you try to insert the (0,0) overlay card using an $END followed by a $FORTRAN, the translator will write utility routines on the output file upon finding the $END. Then it will insert the overlay card, which of course will not now be the first card on the file.

Note that if you wish to insert an overlay card and also insert a PROGRAM card of your own, you must use the following sequence (the $CONSTRAINTS card and statements are optional):
When overlaying a program, remember that each overlay must have a transfer address, which means there must be exactly one PAL $PROGRAM in each overlay. In order to effect a return to a calling overlay, use a >GO TO which transfers control to the end of the PROGRAM. For this purpose, you may wish to make the last statement in the program an identifier. Then a >GO TO this identifier will terminate execution of the overlay and return to the calling overlay.

**Loading Order**

The remaining problem is to get all of the binary decks loaded in the correct order. This can become a problem because the PAL library routines must be part of the main overlay.

If the entire PAL program is to be in the main overlay, this is not a problem. First you load the PAL program binary file (which begins with the OVERLAY (0,0) card and contains no other overlay cards). Next, load the PAL library routines, thus making them part of the main overlay. Last, load the model binary file, which will contain the OVERLAY (1,0) and other overlay cards.

If the PAL program itself consists of more than one overlay, there are two different ways you can get the PAL library routines into the correct position. One method is to take the conversation binary file, the library routines, and the model, and combine them all into one file,
using copy routines. The only restriction on the order is that the library routines must come somewhere after the first two routines (SZZPAL and SZZDAT) and before the next overlay card on the conversation file. This will result in one large file containing everything necessary for a run. The disadvantage of this is that you must either recreate this file from its parts each time you run, or you must maintain a larger file than necessary (larger, because it contains an "extra" copy of the library routines).

The alternative is to split the conversation binary file into two parts which will be loaded separately. The logical place for the break would be at the end of the main overlay, but it could be done at any point after the first two subroutines (SZZPAL and SZZDAT) and before the next OVERLAY card. In order to execute it, you would first load the first part of the conversation, next load the library routines, then load the remainder of the conversation, followed by the model. If desired, the second portion of the conversation and the model binary could be combined into one file, thus reducing the number of files required.

**Running the Model**

Using overlays imposes a restriction on the use of the >RUN command. The >RUN command may be used only in the main overlay. If the entire PAL program is in the main overlay, this is not a problem. If the PAL program consists of more than one overlay, then any $PROGRAM or $SUBROUTINE in a lower overlay must not contain a >RUN command. The PAL translator does not check for this, so you must be careful when writing the program.

This restriction comes about because of the way one overlay replaces another one in core. Suppose a >RUN command were encountered during
execution of a lower level overlay. This results in a call to a utility routine (SZZRUN) which is always in the (0,0) overlay. This routine calls in the model overlay which will replace the conversation overlay in which the >RUN was found. When the model is finished, it will return control to SZZRUN which will attempt to return to the conversation. But the conversation overlay will no longer be there, and the results at that point are unpredictable. The conversation will not resume and the program will probably abort.

This same problem exists for a >RUN command which is entered from the terminal. If some portion of the conversation other than the main overlay is being executed when the >RUN command is entered, it may be impossible to return to the conversation. Some safeguards are built into the PAL library to minimize this problem. The library routines cannot differentiate between overlayed and nonoverlayed programs, so these restrictions apply to all programs. First of all, it is assumed that a user who enters a >RUN command is usually not interested in returning to the conversation. As a result, the user will be required to switch to command mode before using a >RUN command. This will mean that after the model is finished, it will not attempt to return to the conversation. If he is in command mode and he enters a >RUN command, he will be warned that, if he later attempts to change back to a conversation mode, the program may abort. After the warning message is printed, the model will be run for him.
APPENDIX A
USING PAL ON THE CDC 6500

Compiling a PAL Program

The PAL translator is available on a permanent file called PAL. The translator requires about 52000 words of central memory. This does not vary with the size of the program being translated. For a 600-card PAL program, central processor time required is about 15 seconds.

The translator uses a total of nine local files: INPUT, OUTPUT, PALFTN, PALMSG, FZZCOM, FZZDIM, FZZEQU, FZZDAT, and FZZWRK. The PAL program is read from INPUT. A listing of the PAL program and error messages will be on OUTPUT. The FORTRAN code produced by the translator will be on PALFTN. PALMSG is a message file for interactive use (see below). The five remaining files are scratch files used for intermediate code and are of no value to the user.

Immediately following the source listing on OUTPUT will be a table of the variables declared in the program. Each line of the table will have a variable name in the first column, followed by the characteristics of that variable. The second column is the type of the variable: REAL, INTEGER, or HOLLERITH. The third column will indicate if a variable is LOCAL or INITIAL. A local variable should not be declared INITIAL; if it is, it will be identified only as LOCAL. The remaining columns will have the dimensions of the variable if it is an array.

After the variable table will be two lines with information about the efficiency of a hashing function, which the PAL translator uses. The efficiency of the function depends on the variable names used in the PAL program. If any errors were found in the program, the number of errors will be printed at the end of the listing. If no message about errors appears, then no errors were found.
The FORTRAN code on PALFTN must be compiled by the FORTRAN compiler. This can be done in the same job as the PAL translation. If there are fatal errors in the PAL translation, PALFTN will begin with an end-of-file, so the FORTRAN compiler will not compile it.

To compile from a batch job you will need the following control cards:

```
ATTACH,PAL,PAL.
PAL.
FTN,I=PALFTN.
CATALOG,LGO,...
```

This would be followed by an end-of-file and your PAL deck. If your program is on a file, just put the local file name on the PAL card:

```
PAL, lfn.
```

The translator can also be used interactively. In this case, you will usually want to disconnect OUTPUT before starting, or else the entire PAL program will be listed at the terminal. A special message file, PALMSG, has been provided for interactive use. If you connect PALMSG before beginning translation, messages will be printed when the translator begins each new section of the program. If there are any fatal or nonfatal errors, a count of the number of errors will be printed. If no message about errors is printed, then there were no errors found.

**Executing a PAL Program**

You will generally need four files to run a conversation. One of these files will contain the compiled PAL conversation. This file must be the first file loaded because it establishes some of the common block sizes. You also need the file with the compiled FORTRAN model. The third file contains the PAL library routines and is available as permanent file PALLIB.
The standard values are read from unit 7. This means that the standard value file should be attached to TAPE7. An alternative method would be to use another local file name and put it on your execution card. The partial sequence of cards might be:

ATTACH,STDVAL,STDVALUEFILE.
LOAD,CONVER.
LOAD,MODEL.
PALLIB,STDVAL.

The core required for the library routines is about 7000₃ words. The size of the compiled PAL program can vary widely, of course, but it will be roughly 1000₀₃ to 3000₀₃ words. Long text statements, explanation fields, and constraint statements probably contribute the most to the length of a program.
APPENDIX B

FORTRAN CODE GENERATED BY THE PAL TRANSLATOR

The PAL translator translates a PAL program into a set of FORTRAN routines. There are six FORTRAN subroutines produced from the declarative sections. Following them are the routines produced from the conversation sections. Each PAL $PROGRAM or $SUBROUTINE produces two FORTRAN routines.

Declarative Sections

The first six subroutines that are generated are:

1. SZZPAL
2. SZZDAT
3. SZZINC
4. SZZINR
5. SZZRUN
6. SZZCNS

All six routines are always written and they are always in the above order.

SZZPAL is a BLOCK DATA routine which contains the common blocks from the PAL $COMMON sections. Since this subroutine is always first, it establishes the sizes of the common blocks. It also contains DATA statements containing initial values for local variables.

SZZDAT is a BLOCK DATA routine which establishes tables that have information about the variables in the program. These tables are used by the PAL library routines.

SZZINC and SZZINR are interface routines. Sometimes calls must be made from the conversation to the PAL library routines and additional
information must be supplied. The interface routines provide this information.

SZZRUN is responsible for executing the model.

SZZCNS is generated from the $CONSTRAINTS section. It does the actual checking of the values against the constraints. If no $CONSTRAINTS section exists, SZZCNS will be an empty subroutine.

**Conversation Sections**

Two FORTRAN routines are produced for each PAL section. The first routine will be a PROGRAM or SUBROUTINE, depending on whether the PAL section was a $PROGRAM or $SUBROUTINE. It will contain the code for executing the conversation. Its name will be the same as that of the PAL section. The second routine will be a BLOCK DATA routine. It will contain DATA statements for storing the text found in the PAL section. The BLOCK DATA routine generated by the first conversation section will be named SZZ10. The BLOCK DATA for the second conversation section will be SZZ20, the third will be SZZ30, etc., up to SZZ990.

If the PAL program contains no conversation sections, a main FORTRAN program will be generated with the name ZZZZ. It will be set up to read the standard values into the command blocks and place the program in command mode. When it is executed, an arrow will be printed at the terminal and the user can then proceed to enter commands.