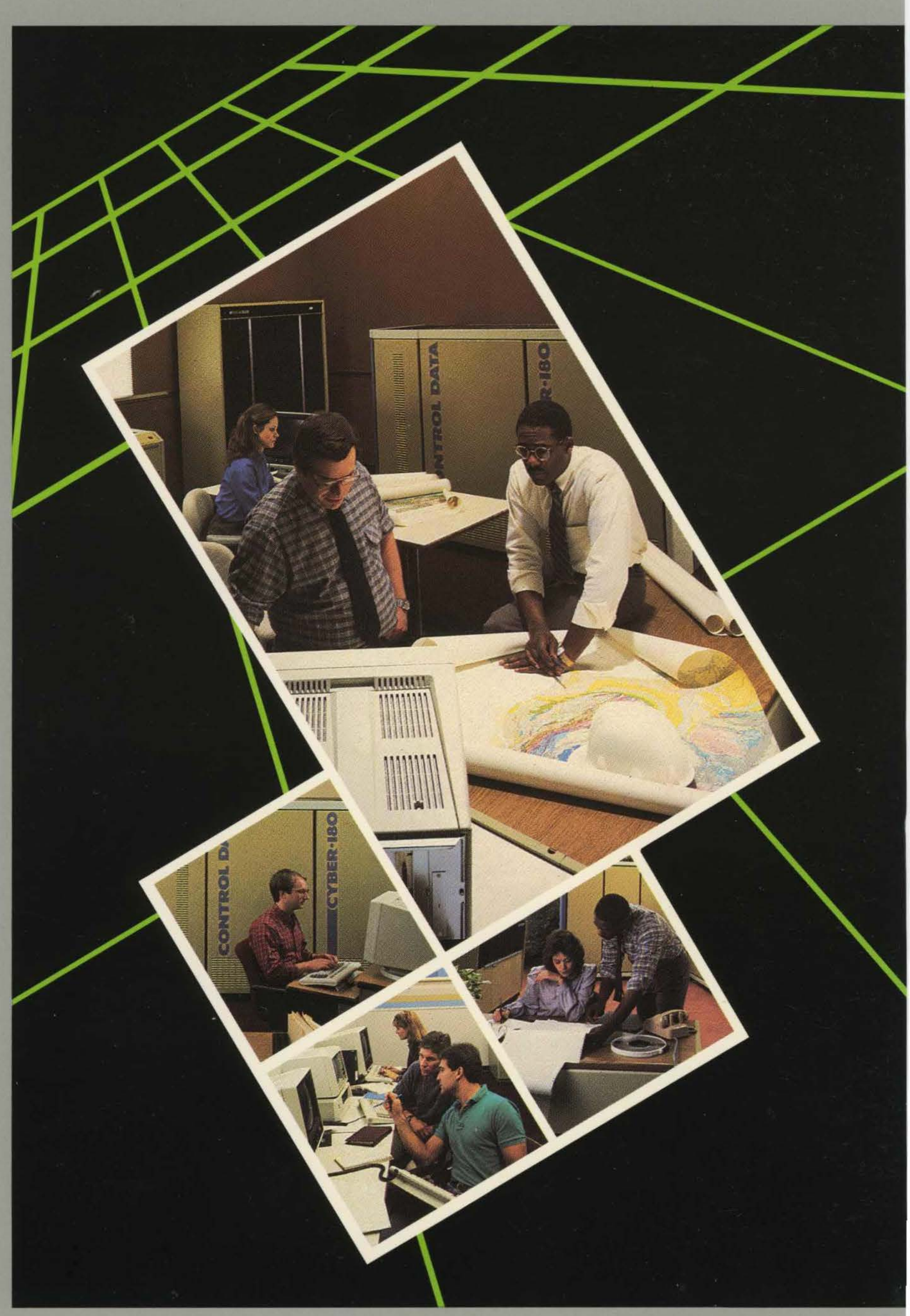
NOS/VE

Screen Formatting

Usage



60488813

CONTROL DATA

NOS/VE

Screen Formatting

Usage

This product is intended for use only as described in this document. Control Data cannot be responsible for the proper functioning of undescribed features and parameters.

Publication Number 60488813

Manual History

Revision	System Version	PSR Level	Product Version	Date
A	1.2.1	670	1.0	December 1986
B	1.2.2	678	1.1	April 1987
C	1.3.1	700	4.0	April 1988
D	1.4.1	716	4.0	December 1988

This manual reflects the release of Screen Formatting under NOS/VE Version 1.4.1, PSR Level 716.

This revision documents the following new features for managing forms:

- The MANAGE_FORMS utility can be used both from within SCL procedures and interactively to display forms. See chapter 2.
- Pascal procedures are available to manage forms. See chapter 5.

Some of the information in this manual is reorganized and rewritten. Change bars mark only new information and technical changes.

This edition obsoletes all previous editions.

©1986, 1987, 1988 by Control Data Corporation
All rights reserved.
Printed in the United States of America.

Contents

About This Manual 5

- Audience 5
- The NOS/VE User Manual Set 6
- Conventions 8
- Submitting Comments 9
- CYBER Software Support Hotline 9

Introduction to Screen Formatting 1-1

- What Is Screen Formatting? 1-1
- Example of Creating and Managing a Form 1-6
- Coordinating Tasks Using a Design Specification 1-11
- Screen Formatting Capabilities 1-12

Using SCL Procedures to Manage Forms 2-1

- Creating an SCL Procedure to Use Forms 2-3
- Storing the Procedure in an Object Library 2-28
- Helping the User Start the Application 2-28
- Options for Managing Forms 2-30
- MANAGE_FORMS Command and Subcommands for Interacting with Forms .. 2-36
- MANAGE_FORMS Functions 2-63

Using COBOL to Manage Forms 3-1

- Writing a Program to Use Forms 3-2
- Expanding and Compiling a Program ... 3-30
- Helping the User Start the Application 3-32
- COBOL Subroutine Calls for Interacting with Forms 3-35

Using FORTRAN to Manage Forms 4-1

- Writing a Program to Use Forms 4-2
- Expanding and Compiling a Program ... 4-24
- Helping the User Start the Application 4-26
- FORTRAN Subroutine Calls for Interacting with Forms 4-29

Using Pascal to Manage Forms 5-1

- Writing a Program to Use Forms 5-2
- Expanding and Compiling a Program ... 5-27
- Helping the User Start the Application 5-29
- Pascal Procedure Calls for Interacting with Forms 5-32

Using CYBIL to Manage Forms 6-1

Writing a Program to Use Forms............... 6-2

Expanding and Compiling a Program.... 6-25

Helping the User Start the Application.......... 6-27

CYBIL Procedure Calls for Interacting with Forms 6-30

Using CYBIL to Create Forms..................... 7-1

More About Forms 7-2

How to Create a Form .. 7-19

The Design Specification . 7-20

Instructions for Designing Forms........ 7-22

Rectangle Form Program................ 7-34

Creating Form Definition Records for Existing Forms 7-42

Attributes for a Form ... 7-43

CYBIL Screen Formatting Procedures.. 7-85

Glossary A-1

Related Manuals B-1

Ordering Printed Manuals................ B-1

Accessing Online Manuals................ B-1

Screen Formatting and Terminal Definitions..... C-1

COBOL Parameter Definitions................ D-1

Pascal Status Constants .. E-1

CYBIL Constants and Types..................... F-1

Constants F-1

Types F-3

FORTRAN Call Definitions................ G-1

Accessing Online Examples................. H-1

Accessing Examples by Name or by Manual..... H-2

Searching for Examples by Command or Procedure Name H-3

Viewing, Copying, Printing, and Executing Examples............... H-4

Using Function Keys and Prompts............ H-5

Index Index-1

About This Manual

This manual describes the CONTROL DATA® Screen Formatting application for use under the CDC® Network Operating System/Virtual Environment (NOS/VE).

Audience

The first chapter of this manual describes Screen Formatting in a manner that does not require knowledge of programming.

The remainder of this manual is directed to application programmers who want to create forms with CYBIL programs and manage them by writing SCL procedures or COBOL, FORTRAN, Pascal, or CYBIL programs that use Screen Formatting. You need knowledge of these programming languages, as well as some knowledge of NOS/VE and the System Command Language (SCL) as presented in the Introduction to NOS/VE manual.

The NOS/VE Screen Design Facility manual describes a screen interface you can use for creating forms using Screen Formatting that requires no programming knowledge.

The NOS/VE User Manual Set

This manual is part of a set of user manuals that describe the command interface to NOS/VE. The descriptions of these manuals follow:

Introduction to NOS/VE

Introduces NOS/VE and SCL to users who have no previous experience with them. It describes, in tutorial style, the basic concepts of NOS/VE: creating and using files and catalogs of files, executing and debugging programs, submitting jobs, and getting help online.

The manual describes the conventions followed by all NOS/VE commands and parameters, and lists many of the major commands, products, and utilities available on NOS/VE.

NOS/VE System Usage

Describes the command interface to NOS/VE using the SCL language. It describes the complete SCL language specification, including language elements, expressions, variables, command stream structuring, and procedure creation. It also describes system access, interactive processing, access to online documentation, file and catalog management, job management, tape management, and terminal attributes.

NOS/VE File Editor

Describes the EDIT_FILE utility used to edit NOS/VE files and decks. The manual has basic and advanced chapters describing common uses of the utility, including creating files, copying lines, moving text, editing more than one file at a time, and creating editor procedures. It also contains descriptions of subcommands, functions, and terminals.

NOS/VE Source Code Management

Describes the SOURCE_CODE_UTILITY, a development tool used to organize and maintain libraries of ASCII source code. Topics include deck editing and extraction, conditional text expansion, modification state constraints, and using the EDIT_FILE utility.

NOS/VE Object Code Management

Describes the CREATE_OBJECT_LIBRARY utility used to store and manipulate units of object code within NOS/VE. Program execution is described in detail. Topics include loading a program,

program attributes, object files and modules, message module capabilities, code sharing, segment types and binding, ring attributes, and performance options for loading and executing.

NOS/VE Advanced File Management

Describes three file management tools: Sort/Merge, File Management Utility (FMU), and keyed-file utilities. Sort/Merge sorts and merges records; FMU reformats record data; and the keyed-file utilities copy, display, and create keyed files (such as indexed-sequential files).

NOS/VE Terminal Definition

Describes the DEFINE_TERMINAL command and the statements that define terminals for use with full-screen applications (for example, the EDIT_FILE utility).

NOS/VE Commands and Functions

Lists the formats of the commands, functions, and statements described in the NOS/VE user manual set. A format description includes brief explanations of the parameters and an example using the command, function, or statement.

Conventions

The following conventions are used in this manual:

Boldface	In a format, boldface type represents names and required parameters.
Italics	In a format, italic type represents optional parameters.
UPPERCASE	In a format, uppercase letters represent reserved words defined by the system for specific purposes. You must use these words exactly as shown.
lowercase	In a format, lowercase letters represent values you choose.
Blue	In examples of interactive terminal sessions, blue represents user input. In program examples, blue identifies comments.
Vertical bar	A vertical bar in the margin indicates a technical change.
Numbers	All numbers are decimal unless otherwise noted.

Submitting Comments

There is a comment sheet at the back of this manual. You can use it to give us your opinion of the manual's usability, to suggest specific improvements, and to report errors. Mail your comments to:

Control Data
Technical Publications ARH219
4201 North Lexington Avenue
St. Paul, Minnesota 55126-6198

Please indicate whether you would like a response.

If you have access to SOLVER, the Control Data online facility for reporting problems, you can use it to submit comments about the manual. When entering your comments, use NV0 (zero) as the product identifier. Include the name and publication number of the manual.

If you have questions about the packaging and/or distribution of a printed manual, write to:

Control Data
Literature and Distribution Services
308 North Dale Street
St. Paul, Minnesota 55103-2495

or call (612) 292-2101. If you are a Control Data employee, call (612) 292-2100.

CYBER Software Support Hotline

Control Data's CYBER Software Support maintains a hotline to assist you if you have trouble using our products. If you need help not provided in the documentation, or find the product does not perform as described, call us at one of the following numbers. A support analyst will work with you.

From the USA and Canada: (800) 345-9903

From other countries: (612) 851-4131

Introduction to Screen Formatting 1

What Is Screen Formatting? 1-1

Example of Creating and Managing a Form 1-6
  Graphic or Text Objects 1-6
  Variable Text Objects 1-7
  Events 1-10

Coordinating Tasks Using a Design Specification 1-11

Screen Formatting Capabilities 1-12

Introduction to Screen Formatting 1

This chapter explains the NOS/VE Screen Formatting application and gives an example of how to use it.

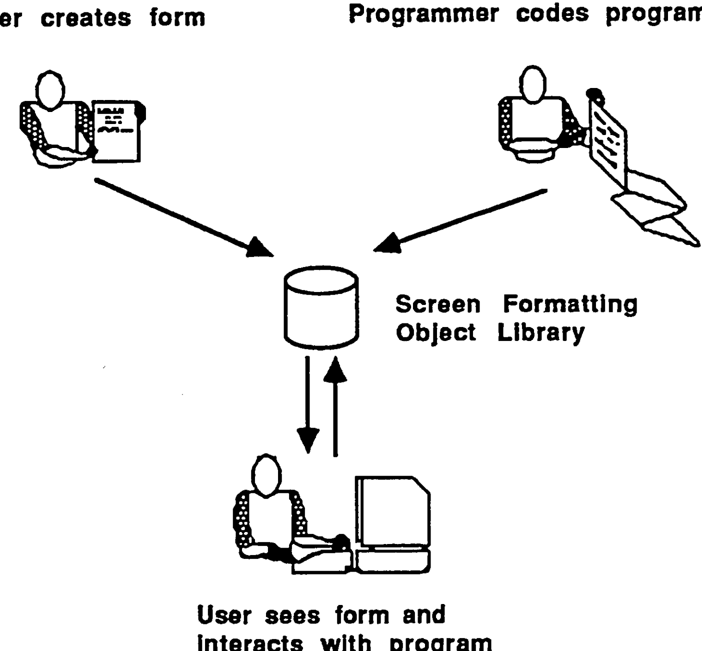
What Is Screen Formatting?

Screen Formatting consists of a set of subroutines and procedures on system object library $SYSTEM.FDF$LIBRARY. Using Screen Formatting subroutines and procedures, you can design a *form* that the user of an application program sees on the screen and uses to interact with the program. For example, for a program that computes the area of circles and rectangles, you might use Screen Formatting to design the following form:

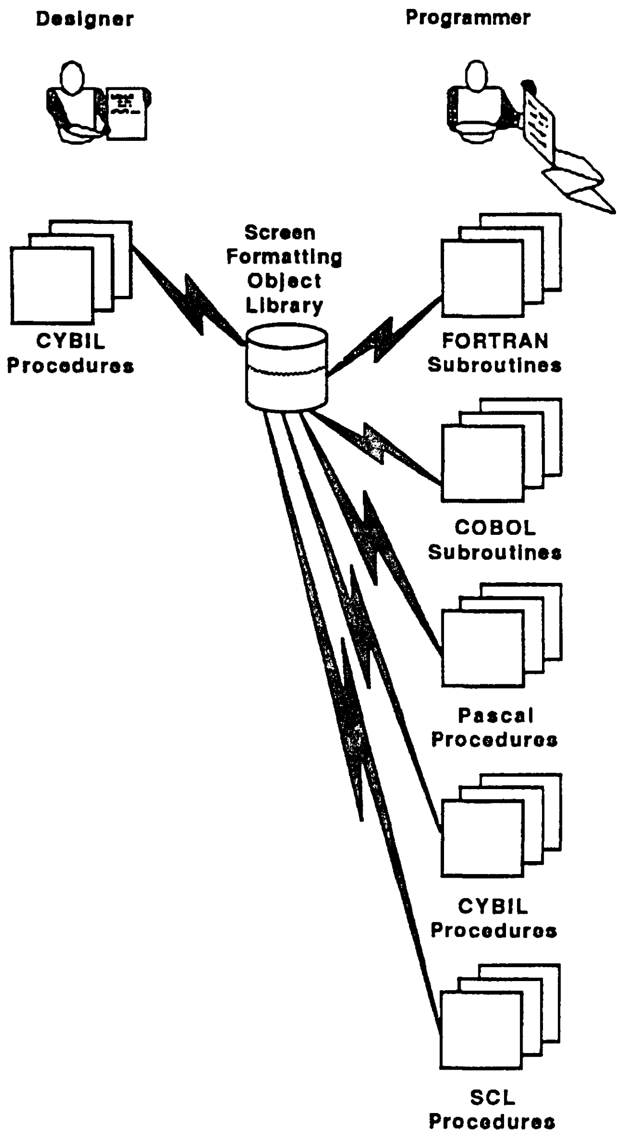
```
Select Object for Computing Area

      Circle
      Rectangle

  Type c or r: _
```

Besides designing the forms, you use Screen Formatting to manage the forms in the application program; for example, you use Screen Formatting to display and remove the forms from the application user's screen.

Designing the forms and managing the forms in the program are separate tasks, usually performed by two people. A *designer* familiar with the needs of the application user creates the forms and puts them on an object library; an *application programmer* manages the forms in the application. When a user executes the application, Screen Formatting combines the work done by the designer and the programmer:

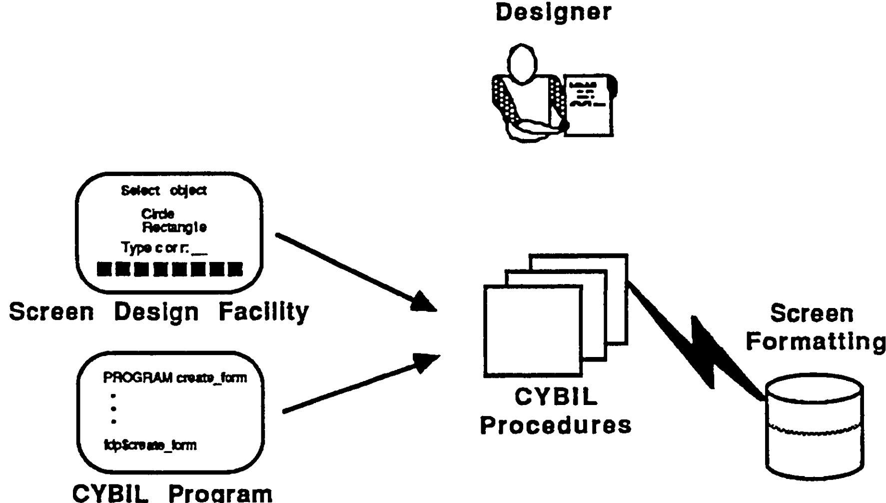


Screen Formatting provides different sets of procedures and subroutines for designing and managing forms. The form designer uses a set of CYBIL procedures, and the application programmer chooses between a set of COBOL subroutines, FORTRAN subroutines, Pascal procedures, CYBIL procedures, or SCL procedures, depending on the language of the application program:



Application programmers access the procedures or subroutines that manage forms by including calls to the procedures or subroutines in the application program.

Designers, on the other hand, have a choice of how to access the CYBIL procedures that create forms. They can either call the procedures in a CYBIL program or use a screen interface provided by the Screen Design Facility:



With the Screen Design Facility, the designer uses function keys to draw the form on the screen, save its image, and define its characteristics. A designer who is not a CYBIL programmer will probably choose this method of designing forms.[1]

Designers who want to either provide special forms for help information or redefine forms while the application is running must use a CYBIL program to create the form. With CYBIL, the form is described in code, using attributes.

1. For more information, see the NOS/VE Screen Design Facility manual.

Screen Formatting also includes subroutines and procedures that relieve the program of some of the tasks it normally performs. For example, for a form that contains a table with more values than can be displayed at one time, Screen Formatting includes procedures and subroutines that page or scroll through the values.

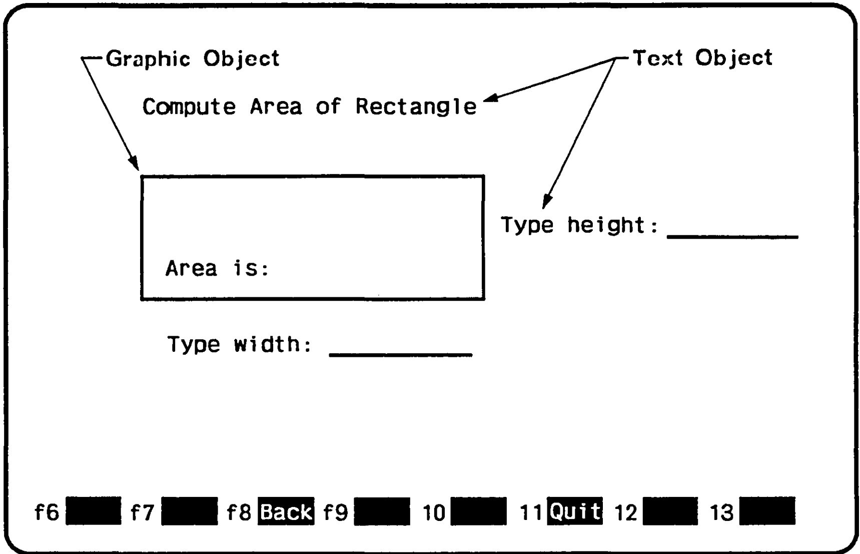
Screen Formatting is an intermediary between a form and the program. This means that when an application user enters a value on a form, the value is sent not to the program, but to Screen Formatting. Screen Formatting determines when to pass the value on to the program by the information it recieves from the program.

Example of Creating and Managing a Form

Using a specific form as an example, this section shows how the form designer and application programmer divide the tasks that create and manage forms.

Graphic or Text Objects

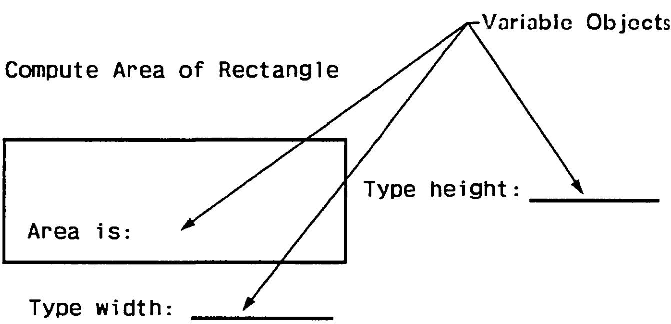
A form contains several discrete areas, each of which Screen Formatting calls either a graphic or a text *object*:



- The designer:
  - Determines what graphic or text objects appear on the form.
  - Defines *display attributes* for the objects. The designer chooses from many different attributes, such as blinking, inverse video, color, or underline.
  - Names the form so the programmer can identify it in the program.
- The programmer displays the form and removes it from the screen using the name assigned by the designer.

Variable Text Objects

For some forms, the designer's and programmer's tasks may be complete as just described. However, the example form has two objects that allow the application user to enter variables and two objects that allow the program to return variables:



Variable text objects require the designer and programmer to perform additional tasks.

- The designer:
  - Defines the text objects to accept variables from the user or the program.
  - Names each text object and display attribute so the programmer can identify them in the program. For this form, the designer:
    - Assigned the name SIDE to the variable text object for the height of the rectangle. The object has a display attribute of underline. (This is the first occurrence of the variable SIDE.)
    - Assigned the name SIDE to the variable text object for the width of the rectangle. The object has a display attribute of underline. (This is the second occurrence of the variable SIDE.)
    - Assigned the name AREA to the variable text object for the computed area. The object has a display attribute of normal (this attribute is the same as the form's attribute).
    - Assigned the name ERROR to the display attribute of inverse video.

- Assigned the name MESSAGE to the variable text object for messages the program displays when user entries are incorrect. The object has a display attribute of normal (this attribute is the same as the form's attribute). Because there is no constant text identifying where the message appears, the object is not identified on the form unless the program moves text to it.

- Defines the types of values the user can enter and the program can return. (On this form, the user can enter integers and the program returns an integer.)

- Defines the action the user takes to send the values to Screen Formatting. (For this form, the designer might define the action as pressing the return key.) An action like this returns control to Screen Formatting and is called an *event.*

- Names the event so the programmer can identify it in the program. (For this form, the event defined as pressing the return key is called COMPUTE.)

- Defines the event as a task that Screen Formatting either performs itself or passes to the program. (For this form, the user enters values for the program to compute, so the designer defines pressing the return key as passing the event from Screen Formatting to the program.)

- The programmer:

  - Controls the position of the cursor, allowing the user to enter data in a specific order.

  - Causes the program to wait for the events the user executes.

  - Provides the code to process events. For the event named COMPUTE on this form, the programmer:

    - Enters calls to Screen Formatting to get the values the user entered for variable text objects from the form to the program. On the call, the programmer specifies the name of the variable text object. (For this form, the name is SIDE.) The programmer then causes the program to go to the part that computes the area.

    - Replaces data on the form using the names of variables defined as objects on the form.

- Includes a call to Screen Formatting to redisplay the screen showing the computed area of the rectangle in the variable text object named AREA.
- Includes calls to Screen Formatting to identify invalid entries to the user. With these calls, the programmer:
  - Positions the cursor at the error.
  - Changes display attribute of the object in error by using the display attribute that was named ERROR by the designer.
  - Displays an error message explaining how to correct the error.

  For example, if the user entered 4.2 as the height of the rectangle, the following is displayed:

Compute Area of Rectangle

Area is:

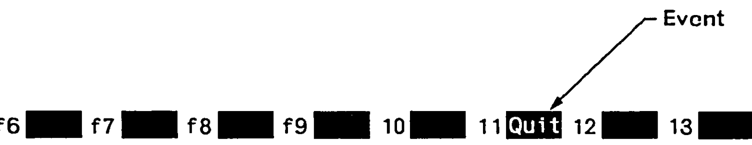
Type height: 4.2

Type width: 3

Type integer value for height

Events

At the bottom of the example form is a menu that contains an event the user can execute by pressing a function key.



The menu is optional and requires the designer and programmer to perform additional tasks.

- The designer:
  - Names the event so the programmer can identify it in the program and defines it to appear as part of a menu of events. (For this form, the name of the event is QUIT and the label appearing on the menu is Quit.)
  - Defines the event as a task that Screen Formatting either performs itself or passes to the program. (For this form, the designer defines the event named QUIT to pass control to the program.)
- The programmer provides code to process the event, identifying it in the program with the name assigned by the designer. (For this form, the programmer defines that the event named QUIT stops the application.)

Coordinating Tasks Using a Design Specification

As you saw in the example, the interaction between the form and the program is complex. To control the process, the designer prepares a list called the *design specification* that tells the programmer what appears on the form and the definitions used for the form and its events. In this specification the designer:

- Names the forms.
- Establishes the order in which forms appear and disappear on the screen.
- Defines and names the variable text objects.
- Defines the types of values the user or program can enter as variables.
- Defines and names the display attributes for objects.
- Defines and names the events that return the user to the program.
- Defines the events that Screen Formatting processes itself.

With this information available, the programmer:

- Displays and removes forms.
- Gets and replaces values on forms.
- Gets and processes events executed by the application user.
- Changes how variable text objects are displayed.
- Changes the position of the cursor on the screen.

Details about these tasks and the formats of the calls the programmer uses are in chapter 2 (for SCL procedure writers), chapter 3 (for COBOL programmers), chapter 4 (for FORTRAN programmers), chapter 5 (for Pascal programmers), and chapter 6 (for CYBIL programmers).

The designer's tasks and the formats of CYBIL procedure calls used to create forms are described in chapter 7. (If you want to design forms using the Screen Design Facility, see the NOS/VE Screen Design Facility manual instead.)

Screen Formatting Capabilities

Screen Formatting is capable of performing many tasks for you. By using Screen Formatting you can:

- Increase the communication between the application and the application user. The forms you create can:
  - Use labels to identify user input.
  - Use the terminal's Tabbing capability to move the user quickly from one entry to the next.
  - Access help and error information easily.
  - Quickly execute complete actions by pressing function keys.
  - Identify important areas on the screen using display attributes.
  - Group items on the screen for easy recognition by the user.
- Increase the productivity of the application programmer. Screen Formatting:
  - Provides data conversion both to and from the program.
  - Has special formats for things like currency and upper/lowercase characters.
  - Validates data according to what is defined on the form.
  - Provides default help and error forms that can be tailored.
  - Automatically pages or scrolls data that cannot be displayed on the screen at one time.

- Release the programmer from writing terminal-dependent code. The forms created can be used at more than one terminal. Screen Formatting:

  - Manages the display of objects on forms. The programmer does not need to keep track of where variables or constant text appear on the user's screen. The programmer uses the name of the object in the program.

  - Displays the menu of events defined for the form.

  - Returns defined events to the program.

- Manage data effectively. Screen Formatting can:

  - Display multiple forms at the same time, allowing the user to move from one form to the other by moving the cursor.

  - Move values from the program to the form.

  - Get values from the form to the program.

Using SCL Procedures to Manage Forms 2

Creating an SCL Procedure to Use Forms 2-3
  Calling Screen Formatting 2-3
    Starting and Stopping the MANAGE_FORMS Utility 2-3
    Displaying and Removing Forms and Variable Data 2-4
    Processing Events and Data 2-6
      Processing Normal Events 2-6
      Processing Abnormal Events 2-7
  An Example Prototype 2-8
    Getting Ready to Start the Prototype 2-8
      Ensuring the Proper Terminal Environment 2-9
      Making Forms Available 2-9
      Using the Design Specification 2-9
      Capturing Prototype Entries 2-11
    Starting the Prototype 2-11
    Opening the Select, Rectangle, and Circle Forms 2-12
    Displaying the Select Form 2-12
    Interacting with the Select Form 2-13
    Displaying the Rectangle Form 2-13
    Interacting with the Rectangle Form 2-14
    Redisplaying the Select Form 2-15
    Interacting with the Select Form Again 2-16
    Displaying the Circle Form 2-16
    Interacting with the Circle Form 2-17
    Stopping the Prototype 2-18
  Writing the Procedure 2-19
    Using the Design Specification 2-21
    Example SCL Procedure 2-22

Storing the Procedure in an Object Library 2-28

Helping the User Start the Application 2-28
  Setting Up the User's Terminal Environment 2-29
  Selecting a Natural Language 2-30
  Starting the Application 2-30

Options for Managing Forms 2-30
  Creating Variables for the Form 2-31
    Creating One Variable that is a Record 2-31
    Creating Individual Variables 2-32
    Creating Variables Manually 2-33
      Using Screen Formatting to Generate TYPE Declarations . 2-34
  Transferring Variables To and From a Form 2-34
    Automatic Transfer of Variables 2-34
    Manual Transfer of Variables 2-35

MANAGE_FORMS Command and Subcommands for Interacting with Forms . . . 2-36
ADD_FORM . . . 2-37
CHANGE_TABLE_SIZE . . . 2-38
CLOSE_FORM . . . 2-40
COMBINE_FORM . . . 2-41
DELETE_FORM . . . 2-43
GET_FORM_VARIABLE . . . 2-44
MANAGE_FORM . . . 2-46
OPEN_FORM . . . 2-48
POP_FORMS . . . 2-49
POSITION_FORM . . . 2-50
PUSH_FORMS . . . 2-52
QUIT . . . 2-53
READ_FORMS . . . 2-54
REPLACE_FORM_VARIABLE . . . 2-55
RESET_FORM . . . 2-57
SET_CURSOR_POSITION . . . 2-58
SET_OBJECT_ATTRIBUTE . . . 2-60
SHOW_FORMS . . . 2-62

MANAGE_FORMS Functions . . . 2-63
$EVENT_NAME . . . 2-64
$EVENT_NORMAL . . . 2-65
$EVENT_POSITION . . . 2-66

Using SCL Procedures to Manage Forms 2

Chapter 1 presented an example of creating and managing forms. It demonstrated that both the designer and the programmer have specific tasks to accomplish. When creating and managing forms using an SCL procedure, the following tasks must be accomplished:

1. The form designer and programmer plan the forms and procedure.

2. The form designer creates the forms specifying SCL as the form processor (or programming language) and prepares a design specification.

3. The form designer puts the forms in an object library.

4. The programmer runs a prototype of the application to view the forms and check the logic of the application.

5. The programmer codes the procedure, including calls to Screen Formatting SCL procedures based on the design specification. These calls manage the forms created by the designer.

6. The programmer stores the procedure in an object library, writes a user procedure to start the application, and helps the user set up the correct terminal environment for using the forms.

When the last task is complete, the procedure and forms are ready for the application user.

Chapter 2 describes the tasks performed by the programmer and shows them being executed in a SCL procedure. At the end of the chapter you will find format and parameter descriptions for each call to SCL procedures used by Screen Formatting. To use these procedures effectively, you should be familiar with writing SCL procedures (see the NOS/VE System Usage manual).

Although you can manage forms using other languages (as described in other chapters in this manual), you may want to use SCL procedures for the following reasons:

- Generally, it takes less code to complete the same task.
- You can display forms shortly after they are created.
- You can let Screen Formatting manage the variables on the forms.

However, because users are interacting with NOS/VE at the command level, the application will use more resources and may execute slower than it would if it was coded using one of the other languages.

The designer's tasks and the formats of the CYBIL procedure calls that create forms are described in chapter 7. (For information about designing forms using the Screen Design Facility, see the NOS/VE Screen Design Facility manual instead.)

Creating an SCL Procedure to Use Forms

When creating a procedure to use forms, you must:

- Plan the procedure with the help of the design specifications created by the form designer. This step is described in chapter 1.
- Call Screen Formatting interactively to simulate how the procedure will run as an application. This is known as the application prototype.
- Write the actual procedure using the same calls to Screen Formatting made in simulating the application.

The next section describes how to call Screen Formatting in both the prototype and procedure. Sections that follow show you how to create a prototype of an application that uses three forms and how to write a procedure that becomes, when executed, the application the user sees.

Calling Screen Formatting

When using forms in a prototype or in a procedure, you perform four basic tasks:

- Starting the MANAGE_FORMS utility. Within this utility you make calls to Screen Formatting using subcommands.
- Displaying and removing forms and variable data on the application user's screen.
- Processing events executed by the user.
- Stopping the MANAGE_FORMS utility.

Starting and Stopping the MANAGE_FORMS Utility

Start the MANAGE_FORMS utility by entering the NOS/VE command MANAGE_FORMS. Starting the utility gives you access to the MANAGE_FORMS subcommands. These subcommands allow you to display and remove forms and variable data.

Stop the MANAGE_FORMS utility by entering the QUIT subcommand. NOS/VE releases the resources allocated to Screen Formatting and returns any forms you have not closed.

(For the format of the commands that start and stop the utility, see *MANAGE_FORMS* and *QUIT* later in this chapter).

Displaying and Removing Forms and Variable Data

To control the display of forms and variable data on the user's screen, perform the following steps in the given sequence:

1. Open the form.

   When you open a form, Screen Formatting locates it and allocates resources for processing Screen Formatting subcommands that use the form. Screen Formatting automatically creates any variables used on the form that you had not created before opening it.

   Using the MANAGE_FORMS utility, you can open any form created by either the Screen Design Facility (SDF) or a CYBIL program that uses Screen Formatting. The forms can be created for use with any of the following programming languages: COBOL, CYBIL, FORTRAN and Pascal, or can be created for use with SCL procedures. If the forms will be used only in applications runing SCL procedures, we recommend that the form designer create forms specifically specifying SCL as the form processor. At this time, SDF does not allow the designer to specify SCL procedure as a programming language (form processor). Instead, have the designer select CYBIL when creating a form.

   You need open a form only once, no matter how many times you use or update it. For this reason, begin a procedure by opening all the forms you will use. When a form requires a large amount of storage for variables, however, you may want to open that one only when the application user needs it.

   (For the format of the subcommand that opens forms, see *OPEN_FORM* later in this chapter).

2. Add the form.

   When you add a form, Screen Formatting schedules it for display on the application user's screen.

   To display more than one form at a time, add all the forms before you display them. Screen Formatting maintains a list of all forms you add. The last form you schedule for display is the top form on the screen. Because forms are opaque, the top form covers other ones appearing in the same area. The cursor position indicates which form is ready for processing.

When the terminal user completes data entry, the cursor position indicates what form Screen Formatting should process. Variables on this form (and any forms combined with this one) are validated and updated. Variables on other forms are not updated or validated.

(For the formats of the subcommands that schedule forms for display, see *ADD_FORM* and *COMBINE_FORMS* later in this chapter.)

3. Read the form.

   When you read a form, Screen Formatting displays all the forms you've added.

   When a form has an event or input variable defined, reading forms also accepts data from the user and displays values returned by SCL.

   (For the format of the subcommand that reads forms, see *READ_FORM* later in this chapter. When none of the forms scheduled for display have an event or input variable defined, you can use a similar subcommand described in *SHOW_FORM* later in this chapter.)

4. Delete the form.

   When you delete a form, Screen Formatting deletes it from the list of forms scheduled for display. The next time you read a form, the deleted form is removed from the screen. However, the form remains available for later use (you must reschedule it for display).

   (For the format of the subcommand that deletes a form, see *DELETE_FORM* later in this chapter.)

5. Close the form.

   When you close a form, Screen Formatting releases the resources the form uses, and the form is no longer available to the user.

   (For the format of the subcommand that closes a form, see *CLOSE_FORM* later in this chapter.)

Processing Events and Data

When creating a form, the designer defines two types of events a user can execute to return control to your procedure: normal and abnormal.

- For normal events, you perform requested actions such as getting variables, doing computations, and updating the form.
- For abnormal events, you do such things as delete the form and go on, or stop using Screen Formatting.

Processing Normal Events

To process a normal event:

1. Get the name of the event and the position of the cursor from Screen Formatting.

   Screen Formatting validates the data the user enters (the form designer defined the validation rules) and transfers values of screen variables to its storage. The form designer may also have created error forms to be displayed when the user enters an incorrect value or presses a key not defined as an event.

   (For the formats of the functions that get the event name and cursor position, see *$EVENT_NAME* and *$EVENT_POSITION* at the end of this chapter.)

2. Get the data from Screen Formatting storage and transfer it to program storage.

   (For the format of the subcommand that gets data, see *GET_FORM_VARIABLE* later in this chapter.)

3. Replace the data in Screen Formatting storage with the data in program storage.

   (For the format of the subcommand that replace variables, see *REPLACE_FORM_VARIABLE* later in this chapter.)

You can also reset the variables on a form to their original state. (For format of the subcommand that resets variables to their original state, see *RESET_FORM* later in this chapter.)

Processing Abnormal Events

To process an abnormal event:

1. Get the name of the event and the position of the cursor from Screen Formatting.

   Unlike a normal event, Screen Formatting neither validates user entries nor transfers values of screen variables to Screen Formatting storage.

   (For the formats of the functions that get the event name and cursor position, see *$EVENT_NAME* and *$EVENT_POSITION* at the end of this chapter.)

2. Write your own procedure to perform the task the design specification assigns to the event. Typical actions for an abnormal event include:

   - Resetting a form and redisplaying it.
   - Moving the user to a new form for additional processing.
   - Returning the user to a previous form.
   - Stopping the procedure.

   The user's screen is updated upon reading the forms again or ending the procedure.

An Example Prototype

With the prototype you can simulate how the user interacts with forms and what the application must do in response to user actions. Because of the ease in which you can display forms on the screen, you can quickly check the order of the forms the user will see in the application. You can also determine if the forms have the correct presentation on the screen and whether the variables work as described in the design specifications.

In the prototype that follows, the options available to the user on a particular form are mentioned. You will then be asked to act as both the user and the application. As the user, you will enter data and execute events on the form. As the application, you will enter MANAGE_FORMS subcommands and calculate the areas as requested by the user.

The prototype application computes the area of either a rectangle or a circle. To do this, it uses three forms. The first form lets the user choose which area to compute. The other two forms allow the user to enter the required input (either the sides of the rectangle or the radius of the circle) and then display the computed area.

Getting Ready to Start the Prototype

Before starting the prototype, you must:

- Ensure that your terminal environment is set up properly.
- Make the forms you will use accessible to your job.
- Identify the names of all forms and all variables used in the forms and what events can be executed from each form.
- Determine the order of the forms.
- Decide whether you want to capture the entries you make in the prototype.

Using the design specification, you can identify the form and variable names, identify events, as well as determine the order the forms appear. The following sections describe how to accomplish these tasks.

Ensuring the Proper Terminal Environment

To use any screen application your terminal must be identified to NOS/VE; also, characteristics such as the attention character and hold messages should be set. Most application programmers have already established these things in their user prolog. If you have not done this, see Setting Up the User's Terminal Environment later in this chapter.

Making Forms Available

To access the forms, they must be in an object library that is in your command list. Use the CREATE_COMMAND_LIST_ENTRY command to add the object library to your command list. For example, to use the forms in object library $SYSTEM.MANUALS.EXAMPLES_FILES.COMMON_OBJECT_LIBRARY, enter:

```
/create_command_list_entry ..
../entry=$system.manuals.examples_files.common_library
```

The forms used in the prototype that follows are in this library. Once you have entered the preceding command you can access the forms as described in the following sections.

Using the Design Specification

The design specification for the application gives you the names of all forms and variables as well as the order the forms appear. The example prototype uses the following information from the design specification:

- The names for the three forms in the application are:

  SCL_SELECT_FORM
  SCL_RECTANGLE_FORM
  SCL_CIRCLE_FORM

- The application starts by displaying the Select Form (SCL_SELECT_FORM).

- The user can call both the Rectangle Form (SCL_RECTANGLE_FORM) and Circle Form (SCL_CIRCLE_FORM) from the Select Form.

- The following variable text objects are defined on the forms to accept input from the user or to display the computed area:

Variable Object	Description
Select Form:	
OBJECT	Area for user input of *r* or *c*.
Rectangle Form:	
SIDE_TABLE	Table that holds values for the rectangle's sides.
SIDE	Areas (two) for user input of values for the rectangle's sides.
AREA	Area for returning value of computed area.
Circle Form:	
RADIUS	Area for user input of value for the circle's radius.
AREA	Area for returning value of computed area.

- The following events are defined on the forms:

Event	Description
COMPUTE	A normal program event that processes data the user entered on the form. For Select Form, the COMPUTE event checks whether the user entered *r* or *c* and then displays the appropriate form. For the other forms, COMPUTE calculates the area and redisplays the form.
BACK	An abnormal program event that takes the user back to a previous environment. For Select Form, the BACK event stops the program. For the other forms, BACK returns the user to Select Form.
QUIT	An abnormal program event that stops the program.

Capturing Prototype Entries

If you capture the entries you make while running the prototype, you can use them as a starting point in coding the final application. The entries are very helpful if your application is an SCL procedure. If your application consists of programs written in FORTRAN or COBOL, the file containing the entries may not be that helpful.

One way to capture entries is to create a file connection to file $ECHO before starting the prototype. Every entry typed at your terminal is then put in the file you name. After stopping the prototype you can delete the connection that you created and then view the entries. In the following example, file ENTRIES contains your prototype session.

```
/create_file_connection standard_file=$echo file=$user.entries
/manage_forms
mf/   :
mf/quit
/delete_file_connection standard_file=$echo file=$user.entries
/copy_file $user.entries
CI manage_forms
CI    :
CI quit
CI delete_file_connection standard_file=$echo file=$user.entries
```

You can then edit this file, adding the other parts of the program as is necessary.

Starting the Prototype

At the system prompt, enter the MANAGE_FORMS command as follows:

```
/manage_forms variable_evaluation=automatic
mf/
```

You specify the automatic evaluation of variables by including VARIABLE_EVALUATION=AUTOMATIC as a parameter. Screen Formatting will then get and replace all of the variables on the forms displayed at the terminal. When you write the actual procedure for the application, you may choose to get and replace variables yourself. This aspect of Screen Formatting is discussed in *Options for Managing Forms* later in this chapter. Using the option of automatically evaluating variables allows you to enter fewer subcommands while using the prototype.

The MANAGE_FORMS utility operates in line mode in the same manner as other NOS/VE utilities. It returns the prompt *mf/* any time you can enter MANAGE_FORMS subcommands (as well as any NOS/VE commands).

Opening the Select, Rectangle, and Circle Forms

To locate the forms and prepare them for use, enter an OPEN_FORM subcommand for each form:

```
mf/open_form form_name=scl_select_form
mf/open_form form_name=scl_rectangle_form
mf/open_form form_name=scl_circle_form
```

Screen Formatting searches for the forms in the object libraries you have in your command list.

When opening the forms, Screen Formatting creates a variable that is a record for each form. The variables defined on a form become fields in the record. Screen Formatting automatically updates this record whenever there are changes made to the values of variables.

Displaying the Select Form

The user sees the Select Form first. Display the Select Form by entering the following subcommands which add the form name to the list of forms ready for display and then read the list:

```
mf/add_form form_name=scl_select_form
mf/read_forms
```

Displaying forms is a two-step process in order to allow you to add more than one form name to the list before reading the list. This process gives you the capability of displaying more than one form on the screen at the same time.

The following form is displayed:

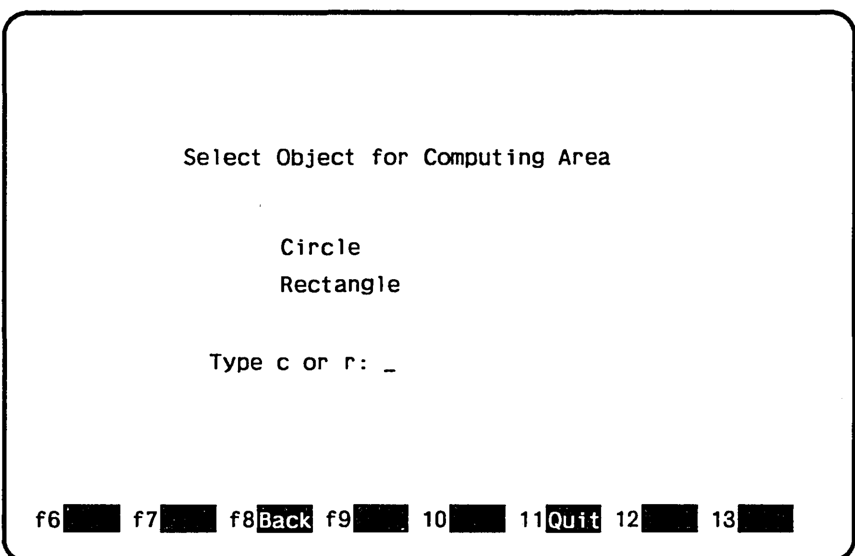


Figure 2-1. Select Form

Interacting with the Select Form

On Select Form, a user enters either *c* to compute the area of a circle or *r* to compute the area of a rectangle. Either the Circle Form or the Rectangle Form will appear next.

Enter *r* and press the return key to go to the Rectangle form. Screen Formatting recognizes pressing the return key as executing the COMPUTE event. It automatically updates the OBJECT variable to contain the letter r and returns the MANAGE_FORMS utility prompt:

```
mf/
```

Displaying the Rectangle Form

Display the Rectangle Form by entering the following subcommands which delete the Select Form from the list of forms scheduled for display, add the Rectangle Form to that list, and read the list:

```
mf/delete_form form_name=scl_select_form
mf/add_form form_name=scl_rectangle_form
mf/read_forms
```

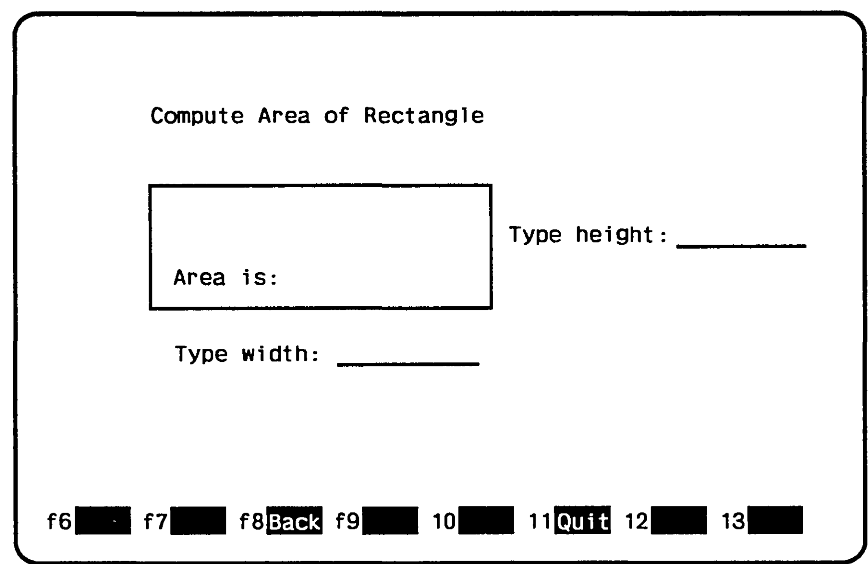


Figure 2-2. Rectangle Form

Interacting with the Rectangle Form

On Rectangle Form, a user enters the lengths of the sides of the rectangle as integers and presses the return key to compute the area.

Enter side lengths of 4 and 5. Then press the return key to execute the COMPUTE event. Screen Formatting sets the first SIDE to 4, the second SIDE to 5, and returns the MANAGE_FORMS utility prompt:

```
mf/
```

To compute the area of the rectangle and then redisplay the Rectangle Form, enter:

```
mf/scl_rectangle_form.area=..
mf../scl_rectangle_form.side_table(1).side*..
mf../scl_rectangle_form.side_table(2).side
mf/read_forms
```

The value of each side is contained in the SIDE fields of the SCL_RECTANGLE_FORM created by Screen Formatting when you opened the form. For more information about how to determine what these fields are see Creating Variables for the Form, later in this chapter.

Screen Formatting updates the AREA variable with the area you computed and then displays the form scheduled for display. Because we did not change the list of forms scheduled for display, the Rectangle Form reappears (showing a value of 20 for the area).

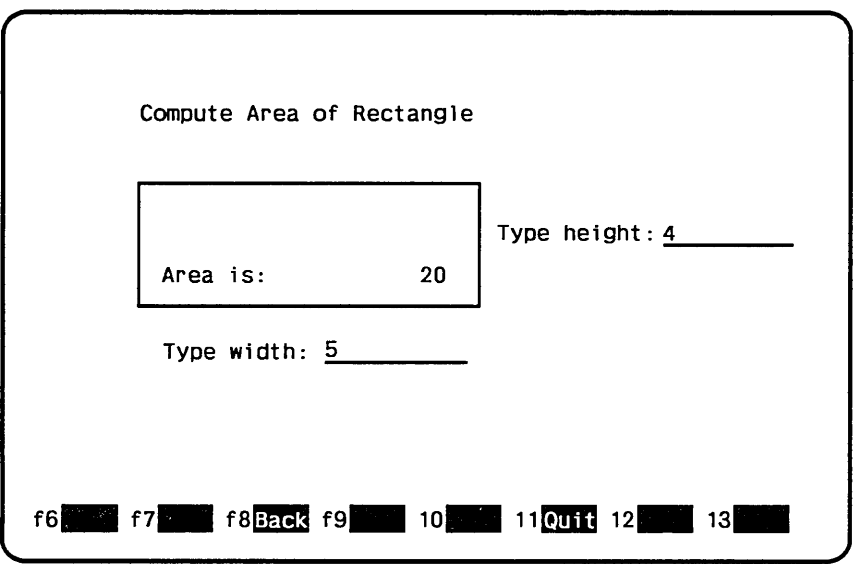


Figure 2-3. Rectangle Form, Showing the Computed Area

A user can also stop the application by executing the QUIT event or can return to the Select Form by executing the BACK event.

Press `Back` to execute the BACK event to return to the Select Form. Screen Formatting returns the MANAGE_FORMS utility prompt:

```
mf/
```

Redisplaying the Select Form

Redisplay the Select Form by entering the following subcommands which delete the Rectangle Form from the list of forms scheduled for display, add the Select Form to that list, and read the list:

```
mf/delete_form form_name=scl_rectangle_form
mf/reset_form form_name=scl_select_form
mf/add_form form_name=scl_select_form
mf/read_forms
```

Screen Formatting deletes the Rectangle Form, resets the Select Form to remove the r from the OBJECT variable, adds the Select Form to the list of forms scheduled for display, and displays the initial Select Form.

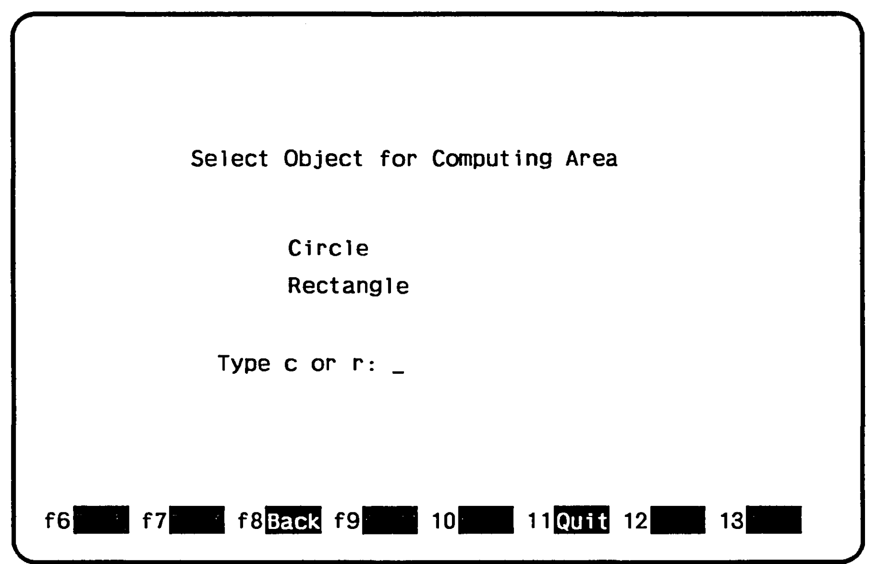


Figure 2-4. Select Form

Interacting with the Select Form Again

Enter *c* and press the return key to go to the Circle Form. Screen Formatting recognizes pressing the return key as executing the COMPUTE event. It automatically updates the OBJECT variable to contain the letter c and returns the MANAGE_FORMS utility prompt:

```
mf/
```

Displaying the Circle Form

Display the Circle Form by entering the following subcommands which delete the Select Form from the list of forms scheduled for display, add the Circle Form to that list, and read the list:

```
mf/delete_form form_name=scl_select_form
mf/add_form form_name=scl_circle_form
mf/read_forms
```

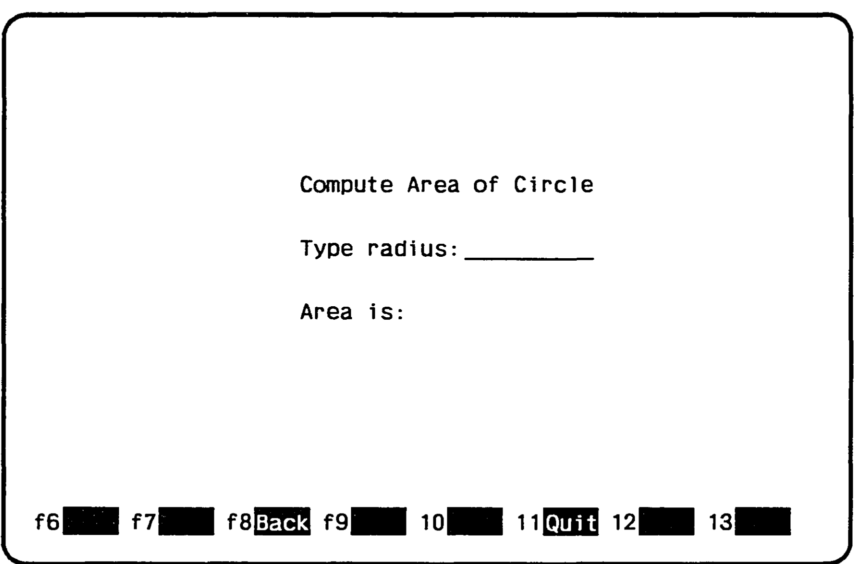


Figure 2-5. Circle Form

Interacting with the Circle Form

On Circle Form, a user enters the length of the radius of the circle as a real value and presses the return key to compute the area.

Enter a radius of 6.8 and press the return key to execute the COMPUTE event. Screen Formatting sets RADIUS to 6.8 and returns the MANAGE_FORMS utility prompt:

```
mf/
```

To compute the area of the circle and then redisplay the Circle Form, enter:

```
mf/scl_circle_form.area=..
mf../3.14*scl_circle_form.radius*scl_circle_form.radius
mf/read_forms
```

Screen Formatting updates the AREA variable with the area you computed and then displays the form scheduled for display. The Circle Form reapears showing a value of 145.194 for the area.

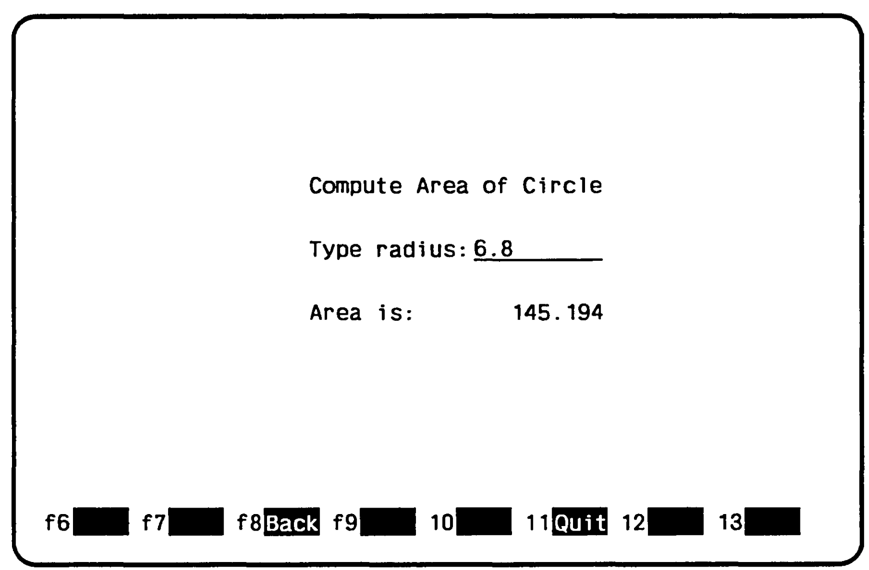


Figure 2-6. Circle Form, Showing the Computed Area

A user can also stop the application by executing the QUIT event or can return to the Select Form by executing the BACK event.

Press `Quit` to execute the QUIT event to stop the application. Screen Formatting returns the MANAGE_FORMS utility prompt:

```
mf/
```

You stop the application by stopping the prototype.

Stopping the Prototype

To stop the prototype, enter:

```
mf/quit
```

Screen Formatting closes all of the forms and NOS/VE releases all resources assigned to Screen Formatting.

Writing the Procedure

Once you have run the prototype and understand how the forms in the application interrelate, you can easily produce the core statements of the procedure. Use the MANAGE_FORMS subcommands entered when you ran the prototype. This ensures that the forms are displayed as they were in the prototype.

In addition to the MANAGE_FORMS subcommands, you need to add the following statements to the procedure:

- The PROCEDURE and PROCEND statements.

  These statements must be present for the procedure to execute.

  (For the formats of PROCEDURE and PROCEND, see the *NOS/VE System Usage* manual.)

- The MANAGE_FORMS command.

  You may want to use different options on the parameters for the command than those used in the prototype. In the prototype, Screen Formatting automatically transfered all variables to and from forms. In the procedure, you can choose to manually transfer variables by using the default of MANUAL for the VARIABLE_EVALUATION parameter on the MANAGE_FORMS command. This allows you to selectively transfer variables to and from forms.

  For more information about the options you have, see *Options for Managing Forms* later in this chapter.

  (For the format of the MANAGE_FORMS command, see its command description later in this chapter.)

- VAR and VAREND statements (to create variables).

  Usually, you create all of the variables used in a procedure at it's beginning. However, with Screen Formatting you have the option of not creating the variables used on the form. When you open a form, Screen Formatting automatically creates the variable needed on that form. You can see all the variables Screen Formatting creates by entering the DISPLAY_VARIABLE_LIST command. It's up to you to create any other variables used in the procedure, such as a status variable.

  (For additional information about creating variables, see *Creating Variables for the Form*, later in this chapter.)

- LOOP, LOOPEND and CYCLE statements (to allow unlimited repetition of processing each form).

  Group the procedure actions according to which form the user is on. Each form can have a number of events that can be executed. Code the procedure's action in response to these events so that each time the user executes the events from the form the same code is processed.

  (For the formats of these statements, see the *NOS/VE System Usage* manual.)

- Error processing statements.

  Screen Formatting automatically validates all user input on a form according to how they were defined on the form. If the status for a particular variable is not normal, your procedure should display an appropriate message on the form.

- GET_FORM_VARIABLE and REPLACE_FORM_VARIABLE subcommands.

  If you choose to transfer variable values to and from the form, you need to add the GET_FORM_VARIABLE and REPLACE_FORM_VARIABLE subcommands to the procedure. When returning the value of a variable to an SCL variable, include the GET_FORM_VARIABLE subcommand. When replacing the value of a variable on a form with a specified value, include the REPLACE_FORM_VARIABLE subcommand.

  (For the format of the GET_FORM_VARIABLE and REPLACE_FORM_VARIABLE subcommands, see their subcommand descriptions later in this chapter.)

Using the Design Specification

While using the prototype of the application, you referred to the design specification to determine the names for the variables on each form and how the forms interrelated. Writing the procedure, you need the part of the design specification that includes the variable objects used for error processing.

The design specification for the example procedure (shown in the next section) lists the variable object used for error processing as MESSAGE. Therefore, you would use the following variables in the procedure:

Variable	Description
SCL_SELECT_FORM.MESSAGE	Area for displaying error messages on the Select Form.
SCL_RECTANGLE_ FORM.MESSAGE	Area for displaying error messages on the Rectangle Form.
SCL_CIRCLE_FORM.MESSAGE	Area for displaying error messages on the Circle Form.

Example SCL Procedure

The following SCL_COMPUTE_AREA procedure uses the forms and design specification described earlier in this chapter. How to execute the procedure is discussed in the sections that follow. You can get a copy of the procedure and execute it using the Examples online manual.

```
PROCEDURE scl_compute_area, sclca (
  status)

  manage_forms
  "$FORMAT=OFF
  VAR
    event: name
    variable_status: status
  VAREND
  "$FORMAT=ON"

" Open  all forms.

  open_form form_name=scl_select_form
  open_form form_name=scl_rectangle_form
  open_form form_name=scl_circle_form

" Add select form to list scheduled for display.

  add_form form_name=scl_select_form

process_selection: ..
  LOOP

" Update screen and accept user terminal entry
" for object; display all added forms.

    read_forms

" Get screen events to determine next actions.

    event=$event_name
```

```
" Stop program on QUIT or BACK event.

    IF (event = 'QUIT') OR (event = 'BACK') THEN
      EXIT process_selection
    IFEND

    get_form_variable form_name=scl_select_form ..
          variable_name=object value=scl_select_form.object

    IF scl_select_form.object = 'R' THEN

"  Compute area of rectangle.

      delete_form form_name=scl_select_form
      reset_form form_name=scl_rectangle_form
      add_form scl_rectangle_form
    process_rectangle: ..
      LOOP

" Update screen with rectangle form.

        read_forms
        event=$event_name
        IF (event = 'QUIT') OR (event = 'BACK') THEN
          delete_form form_name=scl_rectangle_form
          EXIT process_rectangle
        IFEND

" Remove any previous error indications.

        set_object_attribute form_name=scl_rectangle_form ..
              object_name=side attribute=initial
        set_object_attribute form_name=scl_rectangle_form ..
              object_name=side attribute=initial occurrence=2
        scl_rectangle_form.message=''
        replace_form_variable form_name=scl_rectangle_form ..
              variable_name=message ..
              value=scl_rectangle_form.message
```

```
" Get values terminal user entered for sides of rectangle.

       get_form_variable form_name=scl_rectangle_form ..
             variable_name=side occurrence=1 ..
             value=scl_rectangle_form.side_table(1).side ..
             status=variable_status
       IF NOT variable_status.normal THEN
         scl_rectangle_form.message=..
               'Type integer value for height'
         replace_form_variable form_name=scl_rectangle_form ..
               variable_name=message ..
               value=scl_rectangle_form.message
         set_cursor_position form_name=scl_rectangle_form ..
               object_name=side
         set_object_attribute form_name=scl_rectangle_form ..
               object_name=side attribute=error
         CYCLE process_rectangle
       IFEND

       get_form_variable form_name=scl_rectangle_form ..
             variable_name=side occurrence=2 ..
             value=scl_rectangle_form.side_table(2).side ..
             status=variable_status
       IF NOT variable_status.normal THEN
         scl_rectangle_form.message=..
               'Type integer value for width'
         replace_form_variable form_name=scl_rectangle_form ..
               variable_name=message ..
               value=scl_rectangle_form.message
         set_cursor_position form_name=scl_rectangle_form ..
               object_name=side occurrence=2
         set_object_attribute form_name=scl_rectangle_form ..
               object_name=side attribute=error occurrence=2
         CYCLE process_rectangle
       IFEND
```

```
" Compute area of rectangle.

        scl_rectangle_form.area=..
              scl_rectangle_form.side_table(1).side * ..
              scl_rectangle_form.side_table(2).side
        replace_form_variable form_name=scl_rectangle_form ..
              variable_name=area value=scl_rectangle_form.area ..
              status=variable_status
        IF NOT variable_status.normal THEN
          scl_rectangle_form.message=..
                'Format cannot display area'
          replace_form_variable form_name=scl_rectangle_form ..
                variable_name=message ..
                value=scl_rectangle_form.message
        IFEND

      LOOPEND process_rectangle

    ELSEIF scl_select_form.object = 'C' THEN

" Compute area for circle.

      delete_form form_name=scl_select_form
      reset_form form_name=scl_circle_form
      add_form scl_circle_form
    process_circle: ..
      LOOP

"  Update screen with circle form.

        read_forms
        event=$event_name

" On QUIT or BACK event end processing for circle form.

        IF (event = 'QUIT' OR event = 'BACK') THEN
          delete_form form_name=scl_circle_form
          EXIT process_circle
        IFEND
```

```
" Remove any previous error indications.

       set_object_attribute form_name=scl_circle_form ..
             object_name=radius attribute=initial
       scl_circle_form.message=''
       replace_form_variable form_name=scl_circle_form ..
             variable_name=message ..
             value=scl_circle_form.message

" Get terminal user entry for circle radius.

       get_form_variable form_name=scl_circle_form ..
             occurrence=1 variable_name=radius ..
             value=scl_circle_form.radius status=variable_status
       IF NOT variable_status.normal THEN
         scl_circle_form.message=..
               'Type real value for radius'
         replace_form_variable form_name=scl_circle_form ..
               variable_name=message ..
               value=scl_circle_form.message
         set_cursor_position form_name=scl_circle_form ..
               object_name=radius
         set_object_attribute form_name=scl_circle_form ..
               object_name=radius attribute=error
         CYCLE process_circle
       IFEND
```

```
" Compute area of circle.

        scl_circle_form.area=scl_circle_form.radius * ..
              scl_circle_form.radius * 3.14
        replace_form_variable form_name=scl_circle_form ..
              variable_name=area value=scl_circle_form.area ..
              status=variable_status
        IF NOT variable_status.normal THEN
          scl_circle_form.message='Format cannot display area.'
          replace_form_variable form_name=scl_circle_form ..
                variable_name=message ..
                value=scl_circle_form.message
        IFEND

      LOOPEND process_circle

    ELSE

"Terminal user must enter r or c.

      scl_select_form.message='Type r or c.'
      replace_form_variable form_name=scl_select_form ..
            variable_name=message value=scl_select_form.message
      CYCLE process_selection
    IFEND

    IF event = 'QUIT' THEN
      EXIT process_selection
    IFEND

"  Display select form in original state.

    reset_form form_name=scl_select_form
    add_form form_name=scl_select_form
  LOOPEND process_selection
  QUIT

PROCEND scl_compute_area
```

Storing the Procedure in an Object Library

The best way to maintain a created procedure is to put it in an object library. (The object library also contains forms and other procedures you create.) By doing this you can keep track of your procedures and give users easy access to the application.

The following example shows how to place file $USER.SCL_COMPUTE_AREA on object library $USER.EXAMPLE_SOURCE_LIBRARY.

```
/create_object_library
COL/add_module library=$user.example_object_library
COL/combine_module library=$user.scl_compute_area
COL/generate_library library=$user.example_object_library.$next
COL/quit
```

Helping the User Start the Application

The complete application consists of your procedure and the forms created by the designer. To integrate the forms with your program, you must:

- Ensure that the user's terminal environment is set up to use the forms properly (in most instances, by creating a user prolog).
- Ensure that users select the correct natural language.
- Ensure that users know how to start the application.

Setting Up the User's Terminal Environment

To ensure that the user's terminal environment is set up to use the forms properly, set the following terminal characteristics before executing the procedure:

Characteristic	Description
Terminal model	Identifies the terminal to NOS/VE.
Attention character	Provides a character users can enter to interrupt the application. Whenever users have a form on the screen, they must use the attention character to interrupt processing. The suspend (user break 1) and terminate (user break 2) sequences available at the *mfl* prompt do not work when forms are displayed.
Hold messages	Tells the network to hold all network messages until the user stops the application. Otherwise, a computer operator message may overwrite a form while a user is entering data, confusing the user.

In their user prologs, users should set up their terminal for the entire terminal session. The example below does the following:

- Identifies a Digital Equipment Corporation VT220 terminal to the system.
- Chooses the exclamation point as a way to interrupt the program.
- Holds all messages from a NAMVE/CDCNET network.
- Sets up the way the terminal uses the exclamation point to interrupt the program.

The users add the following commands to their user prologs:

```
change_terminal_attributes terminal_model=dec_vt220 ..
  attention_character='!' ..
  status_action=hold
change_term_conn_defaults attention_character_action=1
change_connection_attributes terminal_file_name=input aca=1
change_connection_attributes terminal_file_name=output aca=1
change_connection_attributes terminal_file_name=command aca=1
```

For a further explanation of how to interrupt a screen application during an interactive session and what commands to use for networks other than NAMVE/CDCNET, see the NOS/VE System Usage manual.

Selecting a Natural Language

To ensure that users receive messages in the correct natural language, have them add the CHANGE_NATURAL_LANGUAGE command to their prologs. Because the default language is US_ENGLISH and all messages returned by Screen Formatting are in this language, have users include this command only when you have changed messages to another language.

Changing messages to other languages is described in the NOS/VE Object Code Management manual. The CHANGE_NATURAL_LANGUAGE command is described in the NOS/VE System Usage manual.

Starting the Application

To start the application, enter:

```
/create_command_list_entry e=example_object_library
/scl_compute_area
```

When finished with the application, remove the object library from the command lists:

```
/delete_command_list_entry e=example_object_library
```

Options for Managing Forms

The following Screen Formatting options are available to you:

- Creating variables for the form.
- Transferring variables to and from a form.

Creating Variables for the Form

There are three options available for creating variables. You can:

- Automatically create (with Screen Formatting) one variable for each form you open. The variable is a record containing a field for each variable defined on the form.
- Automatically create (with Screen Formatting) an individual SCL variable for each variable that is defined on each form you open.
- Manually create the variables for each form before you open the form.

Creating One Variable that is a Record

The advantage to creating one variable for each form is that each variable has a unique name (the form name). Therefore, the fields within the variable cannot be confused with fields within another variable (created from another form). For example, the MESSAGE variable on the Rectangle Form (described in the procedure example) is not the same as the MESSAGE variable on the Circle Form (also described in the procedure example).

The disadvantage is that the references to the variables on a form can get long and complex.

To have Screen Formatting automatically create one variable of type RECORD for each form you open, specify VARIABLE_CREATION=FORM_VARIABLE as a parameter on the MANAGE_FORMS command (this is the default). The variable that Screen Formatting creates has the same name as the form.

You can access individual fields within the record by specifying the form name, a period, and the field name. For example, the variable created for the Rectangle Form is SCL_RECTANGLE_FORM. It has three fields corresponding to the three variables defined on the form (SIDE, AREA, and MESSAGE).

You can display the data structure of the variable to determine the field name. For example, after you have opened the Rectangle Form, you can display the data structure of the SCL_RECTANGLE_FORM variable by entering the following SCL command:

```
mf/display_value value=scl_rectangle_form ..
mf../display_options=data_structure
display option: DATA_STRUCTURE

"RECORD"
  SIDE_TABLE: "ARRAY"
                1. "RECORD"
                    SIDE: "INTEGER"  0
                   "RECORD END"

                2. "RECORD"
                    SIDE: "INTEGER"  0
                   "RECORD END"
              "ARRAY END"

   AREA: "INTEGER"  0
   MESSAGE: "STRING"  ''
"RECORD END"
```

Because the fields for both the MESSAGE and AREA variables have no internal structures, you can refer to them as SCL_RECTANGLE_FORM.MESSAGE and SCL_RECTANGLE_FORM.AREA. However, the SIDE variable field is contained in an array named SIDE_TABLE (corresponding to the table created on the form). Within the array is a record for each side of the rectangle. To refer to the side of a rectangle you must include the internal structure of the SIDE_TABLE array. For example, to refer to the first side of the rectangle use:

SCL_RECTANGLE_FORM.SIDE_TABLE(1).SIDE

For more information about the structure of variables that are records, see the NOS/VE System Usage manual.

Creating Individual Variables

The advantage to creating an individual SCL variable for each variable on a form is that the reference to the variable is generally short and corresponds closely to what is defined on the form. For example, the sides of the rectangle defined on the Rectangle Form (described in the procedure example) become elements in the array named SIDE: SIDE(1) and SIDE(2).

The disadvantage to creating individual variables is that variables from different forms can have the same reference. For example, the MESSAGE variable on the Rectangle Form and the MESSAGE variable on the Circle Form (also described in the procedure example) both become MESSAGE.

To have Screen Formatting automatically create individual variables for each opened form, specify VARIABLE_CREATION=SINGLE as a parameter on the MANAGE_FORMS command.

After you have opened a form, you can display the data structure of any individual variable by using the SCL command DISPLAY_VALUE. For example, to display the data structure of the SIDE variable on the Rectangle Form, enter:

```
mf/display_value value=side display_options=data_structure
display option: DATA_STRUCTURE

"ARRAY"
  1: "INTEGER"   0
  2: "INTEGER"   0
"ARRAY END"
```

For more information about the data structure of variables, see the NOS/VE System Usage manual.

Creating Variables Manually

The advantage to creating variables manually is that you can specify different scopes for them. Variables created by Screen Formatting have only a local scope. That is, they are available only within the procedure they were first created.

You create variables using either:

- A TYPE declaration to create variables with a scope of ENVIRONMENT.
- A VAR declaration to create variables with any scope.

Within a CYBIL program, Screen Formatting can generate a form definition record that defines a TYPE for each form. Otherwise, you must create your own TYPE or VAR declarations for variables. See the NOS/VE System Usage manual for details on creating variables.

Using Screen Formatting to Generate TYPE Declarations

The CYBIL Screen Formatting procedure FDP$WRITE_RECORD_DEFINITION generates form definition records for any form you can open. An example of a CYBIL program that generates form definition records is in the online Examples manual under Screen Formatting Examples. For forms used in SCL procedures, a form definition record consists of a TYPE declaration. Each declaration creates one variable that is a record containing all of the variables defined on the form.

To generate TYPE declarations for forms used in SCL procedures, execute the example in the online manual and specify SCL when prompted for the form processor. The TYPE declaration in file created by the example can be copied into the SCL procedure using the forms. Place the declarations before the opening of the forms. Screen Formatting always checks to see if a variable exists and creates only those that do not exist.

For additional information about form definition records, see Creating Form Defintion Records for Existing Forms in chapter 7.

Transferring Variables To and From a Form

There are two options available for transferring variables. You can either:

- Manually transfer variables.
- Have Screen Formatting automatically transfer variables whenever you enter the READ_FORMS or SHOW_FORMS subcommand.

Each of these options has advantages and disadvantages. Manually transferring the values of variables generally uses less system resources because you update only those variables necessary to the action you are performing. Automatically transferring values to variables updates every variable on the form each time you read it. If this is what you would do manually, anyway, then it requires a lot less code to perform it automatically.

Automatic Transfer of Variables

To automatically transfer variables to and from forms, specify VARIABLE_EVALUATION=AUTOMATIC as a parameter on the MANAGE_FORMS command.

Manual Transfer of Variables

To manually transfer variables to and from forms, specify VARIABLE_EVALUATION=MANUAL as a parameter on the MANAGE_FORMS command. This is the default.

Manually transferring the values of variables involves using the GET_FORM_VARIABLE and REPLACE_FORM_VARIABLE subcommands. On these subcommands you identify the variable by specifying the form name, the variable name as defined by the designer, and the number of the occurrence (if the variable is part of a table). For example, when getting the value of the rectangle's height from the Rectangle Form (described in The Application Prototype), use the following GET_FORM_VARIABLE subcommand:

```
mf/get_form_variable form_name=scl_rectangle_form ..
mf../variable_name=side ..
mf../value=scl_rectangle_form.side_table(1).side occurrence=1
```

MANAGE_FORMS Command and Subcommands for Interacting with Forms

The MANAGE_FORMS command and subcommands that follow call Screen Formatting to manage forms. For each subcommand, there is a purpose description, format, list of parameters and their types, and pertinent remarks.

When using the MANAGE_FORMS utility in procedures, you should ensure that the subcommands execute properly. To do this check the value returned in the STATUS and VARIABLE_STATUS (if present) parameters. A list of possible error messages that can be returned can be found in the NOS/VE Diagnostic Messages manual. The product identifier for the messages is FD.

ADD_FORM
MANF Subcommand

Purpose

ADD_FORM schedules a form for display on the application user's screen.

Format

ADD_FORM or
ADDF
  FORM_NAME=data_name
  STATUS=status variable

Parameters

FORM_NAME or FN

The name established when the form was opened. This parameter is required.

Remarks

- When you enter either the READ_FORMS or SHOW_FORMS subcommand, Screen Formatting displays the added form on the terminal screen. The added form is placed on top of other forms occupying the same area on the screen.
- When displayed, each form that is added operates independently from other forms that have been added. When a user executes a normal event, Screen Formatting validates and updates only those variables on the form associated with the event.

  To have forms share events, use the COMBINE_FORM subcommand.
- Before you add a form, you must open it.
- You cannot add a pushed form.

CHANGE_TABLE_SIZE MANF Subcommand

Purpose

CHANGE_TABLE_SIZE changes the size of the table during application execution.

Format

CHANGE_TABLE_SIZE or
CHATS
  FORM_NAME=data_name
  TABLE_NAME=data_name
  TABLE_SIZE=integer
  STATUS=status variable

Parameters

FORM_NAME or FN

The name established when the form was opened. This parameter is required.

TABLE_NAME or TN

The name of the table to change in size. This parameter is required.

TABLE_SIZE or *TS*

The size of the table. While this subcommand is in effect, Screen Formatting limits the number of stored occurrences allowed for a table to the value you specify on this parameter. How many occurrences are displayed at one time depends on the number of visible occurrences defined in the form.

If you specify zero for the table size, no occurrences appear on the form.

Remarks

- The table must be present in an open form.
- The size limitation remains in effect until the next time you enter the CHANGE_TABLE_SIZE subcommand.
- The maximum size for a table is identified by the form as the maximum number of stored occurrences. If you specify a table size larger than the maximum, you receive an error message (fde$invalid_table_size).

Examples

The following examples describe how changing the size of a table affects the application user. On the form, the table's specifications are a maximum of 20 stored occurrences, of which 6 occurrences can be visible at one time.

- If you specify a table size of 10, Screen Formatting displays 6 occurrences and allows the application user to page to the 10th occurrence.
- If you specify a table size of 4, Screen Formatting displays 4 occurrences and does not allow the application user to page.

CLOSE_FORM
MANF Subcommand

Purpose CLOSE_FORM releases resources used to process a form and deletes the form from the list scheduled for display.

Format

CLOSE_FORM or
CLOF
FORM_NAME=data_name
STATUS=status variable

Parameters FORM_NAME or FN

The name established when the form was opened. This parameter is required.

Remarks

- When the you enter either the READ_FORMS or SHOW_FORMS subcommand, Screen Formatting removes the closed form from the terminal screen as a result of entering this subcommand.

- Before you can close a form, you must open it.

- You cannot close a pushed form.

COMBINE_FORM
MANF Subcommand

Purpose

COMBINE_FORM combines a form with a previously added form and schedules the combined form for display on the terminal screen.

Format

COMBINE_FORM or
COMBINE_FORMS or
COMF
ADDED_FORM_NAME=data_name
COMBINE_FORM_NAME=data_name
STATUS=status variable

Parameters

ADDED_FORM_NAME or AFN

The name of the previously added form. This parameter is required.

COMBINE_FORM_NAME or CFN

The name of the form you are combining with the previously added form. This parameter is required.

Remarks

- You cannot combine a pushed form.
- The combined form inherits the event definitions of the previously added form.
- Before you combine a form with a previously added form, you must open both forms.
- When you enter either the READ_FORMS or SHOW_FORMS subcommand, Screen Formatting displays the combined form. The combined form is placed on top of other forms occupying the same area on the screen.
- When you start the MANAGE_FORMS utility specifying VARIABLE_EVALUATION=AUTOMATIC and the application user executes an event to return to the utility normally, Screen Formatting updates all SCL variables associated with both the added and combined forms.

- When you start the MANAGE_FORMS utility specifying VARIABLE_EVALUATION=MANUAL and you enter the REPLACE_FORM_VARIABLE subcommand, Screen Formatting updates the variable on both the added and combined forms.

- To combine several forms with a previously added form, execute this subcommand more than once.

DELETE_FORM
MANF Subcommand

Purpose

DELETE_FORM deletes the form from the list of forms scheduled for display.

Format

DELETE_FORM or
DELF
  FORM_NAME=data_name
  STATUS=status variable

Parameters

FORM_NAME or FN

The name established when the form was opened. This parameter is required.

Remarks

- When you enter either the READ_FORMS or SHOW_FORMS subcommand, Screen Formatting removes the deleted form from the terminal screen and replots any forms uncovered by the deleted form.
- When you add a form (ADD_FORM) again that you previously deleted, the data in the form is retained.
- Before you delete a form, you must open it.
- You cannot delete a pushed form.
- If the form was added and has any combined forms associated with it, the combined forms are also deleted.
- When you delete a combined form, only that form is deleted. Areas covered by the combined form are replotted after the combined form is deleted.

GET_FORM_VARIABLE
MANF Subcommand

Purpose GET_FORM_VARIABLE gets the value the user entered on the form for a variable and transfers it to SCL.

Format **GET_FORM_VARIABLE** or
GETFV
FORM_NAME=data_name
VARIABLE_NAME=data_name
VALUE=any variable
OCCURRENCE=integer
STATUS=status variable

Parameters FORM_NAME or FN

The name of the form where the variable resides. This parameter is required.

VARIABLE_NAME or VN

The name of the variable on the form to get and transfer to SCL. This name was defined when the form was created. This parameter is required.

VALUE or V

Parameter Attributes: BY_NAME

The variable that is to hold the value Screen Formatting gets from the form. This variable is either created automatically when the form is opened or created manually. This parameter is required.

See the VARIABLE_CREATION parameter on the MANAGE_FORM command for more information.

OCCURRENCE or *O*

The occurrence of the variable name. The values allowed are 1 through 1000. Use 1 for the first or only occurrence. The default is 1.

Remarks

- Before you get a variable, you must open its form. If you get the variable after opening the form and before reading or replacing the variable on the form, the utility returns the initial value specified by the form designer.

- If the form designer specifies data validation rules and error processing to display an error message or form, you do not need to look at the STATUS parameter.

  If the form designer specifies data validation rules and no error processing, you must look at the STATUS parameter.

  If the form designer specifies no data validation rules, you must look at the STATUS parameter to determine if the subcommand executed properly.

MANAGE_FORM Command

Purpose MANAGE_FORM begins the MANAGE_FORMS utility. This utility allows you to display and manage forms. A form is a related group of objects shown on a user's terminal screen. Using the MANAGE_FORMS utility, you can display any form created through Screen Formatting.

Format **MANAGE_FORM** or
MANAGE_FORMS or
MANF
VARIABLE_CREATION=keyword
VARIABLE_EVALUATION=keyword
STATUS=status variable

Parameters *VARIABLE_CREATION* or *VC*

The keyword indicating how the utility is to create variables when a form is opened. Use one of the following keywords:

FORM_VARIABLE

Create one variable for each form that is opened. The variable is an SCL record that contains a field for each variable text object on the form. The name of the form becomes the name of the variable. FORM_VARIABLE is the default.

SINGLE

Create one variable for each variable defined on the form. The type of the variable depends on the definition of the variable on the form.

NONE

Create no variables. You are responsible for defining all variables used on the forms you open.

VARIABLE_EVALUATION or *VE*

The keyword indicating how the utility is to evaluate variables. Use one of the following keywords:

AUTOMATIC

Automatic evaluation of variables. Screen Formatting updates all variables from added or combined forms when you execute either READ_FORMS or SHOW_FORMS. You do not need to enter the GET_FORM_VARIABLE and REPLACE_FORM_VARIABLE subcommands.

MANUAL

Manual evaluation of variables. Screen Formatting does not update variables automatically. You must update them using GET_FORM_VARIABLE and REPLACE_FORM_VARIABLE subcommands.

Remarks

- The forms used with the MANAGE_FORMS utility can be created through the Screen Design Facility (SDF) or with a CYBIL program that makes calls to Screen Formatting procedures.
- When executing the utility interactively, the *mf/* prompt is displayed.

OPEN_FORM
MANF Subcommand

Purpose

OPEN_FORM locates a form and prepares it for use by the utility.

Format

OPEN_FORM or
OPEF
  FORM_NAME=data_name
  STATUS=status variable

Parameters

FORM_NAME or FN

The name of the form you want to open. This parameter is required.

Remarks

- When you open forms, Screen Formatting creates SCL variables for form variables. The scope of the variables is LOCAL. Before creating a variable, Screen Formatting checks to see if the variable already exists. If it does, Screen Formatting does not try to create it again.
- Screen Formatting locates a form by searching the command library list to find the form name on the object libraries. (You specify the order in which Screen Formatting searches the list using the NOS/VE command CREATE_COMMAND_LIST_ENTRY).
- Executing OPEN_FORM does not display the form on the screen. (See ADD_FORM, READ_FORMS, or SHOW_FORMS.)

POP_FORMS
MANF Subcommand

Purpose POP_FORMS deletes forms scheduled (added or combined) since the last PUSH_FORMS call.

Format **POP_FORMS** or
POPF
STATUS=status variable

Remarks Events associated with the last list of pushed forms become active.

POSITION_FORM
MANF Subcommand

Purpose POSITION_FORM schedules moving a form to a new location. Using this subcommand, you can define a form at one location and display it at another location, or you can move a form from where it is currently displayed to a new location.

Format **POSITION_FORM** or
POSF
  FORM_NAME=data_name
  X_POSITION=integer
  Y_POSITION=integer
  STATUS=status variable

Parameters FORM_NAME or FN

The form name established when the form was opened. This parameter is required.

X_POSITION or *XP*

The x position on the screen. The character position in the upper left corner of the screen is 1, and the x position increases by 1 for each character counting from left to right. The default is 1.

Y_POSITION or *YP*

The y position on the screen. The character position in the upper left corner of the screen is 1, and the y position increases by 1 for each character counting from top to bottom. The default is 1.

Remarks

- When you enter either the READ_FORMS or SHOW_FORMS subcommand, Screen Formatting displays the form on the screen at the position specified in the POSITION_FORM subcommand.

- If you enter this subcommand while the form is displayed, the form is deleted from its current location and added at the new location. The added form is displayed on top of any other form occupying the same area on the screen.

- If you enter this subcommand before the form is displayed, the form is displayed at the specified location.
- Before you position a form, you must open it.
- You cannot position a pushed form.

PUSH_FORMS
MANF Subcommand

Purpose PUSH_FORMS causes Screen Formatting to record added and combined forms so you can return to them later.

Format **PUSH_FORMS** or
PUSF
STATUS=status variable

Remarks

- Events associated with these forms are not passed to you.
- You cannot change or close a pushed form.
- Pushed forms are displayed on the screen. If you want newly added forms to appear on a blank screen, first add a blank form that covers the screen.

  Updates to the screen continue to show the pushed forms.
- This subcommand deactivates the events associated with forms scheduled for display (added or combined) since the last PUSH_FORMS subcommand.

QUIT
MANF Subcommand

Purpose QUIT ends the MANAGE_FORMS utility session.

Format **QUIT** or
QUI
STATUS=status variable

Remarks All open forms are closed and the resources used by Screen Formatting are returned to NOS/VE.

READ_FORMS MANF Subcommand

Purpose READ_FORMS updates the terminal screen and accepts input from the application user.

Format **READ_FORMS** or
REAF
STATUS=status variable

Remarks

- Executing READ_FORMS:
  - Displays all the forms you scheduled for display and have not deleted. If you added or combined forms since the last READ_FORMS or SHOW_FORMS subcommand, it displays them for the first time.
  - Removes from the screen the forms you deleted since the last READ_FORMS or SHOW_FORMS subcommand.
  - Updates on the screen the variables replaced since the last READ_FORMS or SHOW_FORMS call subcommand.
  - Updates on the screen the objects for which display attributes were set or reset since the last READ_FORMS or SHOW_FORMS subcommand.
- Events not retrieved with the $EVENT_NAME function are deleted before any input is accepted from the user.
- The READ_FORMS subcommand does not execute unless the forms scheduled for display contain at least one active event.
- After issuing this request, you do not regain control until the user issues a normal event and Screen Formatting validates all the data, or the user issues an abnormal event.

REPLACE_FORM_VARIABLE MANF Subcommand

Purpose REPLACE_FORM_VARIABLE transfers an SCL variable to Screen Formatting.

Format **REPLACE_FORM_VARIABLE** or
REPFV
FORM_NAME=data_name
VARIABLE_NAME=data_name
VALUE=any
OCCURRENCE=integer
STATUS=status variable

Parameters FORM_NAME or FN

The name of the form where the variable resides. This parameter is required.

VARIABLE_NAME or VN

The name of the variable to replace. This name was defined when the form was created. This parameter is required.

VALUE or V

The variable that holds the value Screen Formatting will replace on the form. This variable is either created automatically when the form is opened or created manually. This parameter is required.

See the VARIABLE_CREATION parameter on the MANAGE_FORM command for more information.

OCCURRENCE or *O*

The occurrence of the variable name. The values allowed are 1 through 1000. Use 1 for the first or only occurrence.

Remarks

- When you execute either the READ_FORMS or SHOW_FORMS subcommand, Screen Formatting replaces the variable on the terminal screen.
- Before you replace a variable, you must open the form on which it is replaced.
- You cannot replace a variable for a pushed form.
- If the variable is not valid, it is not replaced.

RESET_FORM
MANF Subcommand

Purpose RESET_FORM resets the form to the state specified by the form definition.

Format **RESET_FORM** or
RESF
FORM_NAME=data_name
STATUS=status variable

Parameters FORM_NAME or FN

The name of the form to reset. This parameter is required.

Remarks

- When you execute either the READ_FORMS or SHOW_FORMS subcommand, Screen Formatting displays the form on the terminal screen with the reset specifications.
- All variables belonging to the form have their initial values and display attributes. The form is in its defined position.
- Before you reset a form, you must open it.
- You cannot reset a pushed form.

SET_CURSOR_POSITION MANF Subcommand

Purpose SET_CURSOR_POSITION sets the cursor to a selected position for later display.

Format **SET_CURSOR_POSITION** or
SETCP
  FORM_NAME=data_name
  OBJECT_NAME=data_name
  OCCURRENCE=integer
  CHARACTER_POSITION=integer
  STATUS=status variable

Parameters FORM_NAME or FN

The name established when the form was opened. This parameter is required.

OBJECT_NAME or ON

The name of the object on which you want to set the cursor. This name was defined when the form was created. This parameter is required.

OCCURRENCE or *O*

The integer specifying the occurrence of the object name. Use 1 for the first occurrence. The default is 1.

CHARACTER_POSITION or *CP*

The character position to which you want to set the cursor. Use 1 for the first character position. The default is 1.

Remarks

- One use of this subcommand is to alter the default sequence of the application user's entry of variables. In the default sequence, Screen Formatting places the cursor on the first input variable of the highest priority form. The highest priority form is the form last added, combined, or positioned.

  At terminals with protected fields, the user then tabs from one variable text object to the next. The cursor starts at the top line of the form. It moves from left to right on each line. When no variable text object appears on a line, the cursor moves down to the next line. At terminals without protected fields, the user must move the cursor using the arrow keys or use the tab and return keys,

- When you execute either the READ_FORMS or SHOW_FORMS subcommand, Screen Formatting updates the terminal screen with the cursor at the specified position.

- If the position you specify is not visible on the screen, Screen Formatting shifts the data to make the cursor visible.

- The cursor position is in effect only for the next screen update from reading or showing forms.

- Before you set the cursor position on a form, you must open the form and either add or combine it.

- You cannot set the cursor position in a pushed form.

SET_OBJECT_ATTRIBUTE MANF Subcommand

Purpose SET_OBJECT_ATTRIBUTE changes a display attribute for an object.

Format **SET_OBJECT_ATTRIBUTE** or
SET_OBJECT_ATTRIBUTES or
SETOA
  FORM_NAME=data_name
  OBJECT_NAME=data_name
  ATTRIBUTE=keyword or *data_name*
  OCCURRENCE=integer
  STATUS=status variable

Parameters FORM_NAME or FN

The name of the form containing the object. This parameter is required.

OBJECT_NAME or ON

The name of the object whose display attribute is being reset. This parameter is required.

ATTRIBUTE or *A*

The name given the display attribute being set when the attribute was defined on the form. The attribute used here is defined for the form and not for a specific object. When using Screen Design Facility, screen attributes are defined through the ATTRIB function. When using a CYBIL program, the ADD_DISPLAY_DEFINITION attribute record defines form attributes.

Specifying the keyword INITIAL, resets the object to the attribute defined in the form definition. The default is INITIAL.

OCCURRENCE or *O*

The occurrence of the object. For the first or only occurrence, use 1. The default is 1.

Remarks

- You can set the attributes of objects that are variable text, constant text, lines, or boxes.
- Changed attributes replace existing attributes.
- When you execute either the READ_FORMS or SHOW_FORMS subcommand, Screen Formatting displays the object using the set attributes.
- If the object you specify is not visible on the screen, Screen Formatting shifts the data to make the object visible.
- Before you set the attribute of an object, you must open the form the object is on and either add or combine it.
- You cannot set attributes of objects on a pushed form.

SHOW_FORMS
MANF Subcommand

Purpose SHOW_FORMS updates the terminal screen.

Format **SHOW_FORMS** or
SHOF
STATUS=status variable

Remarks

- When none of the forms scheduled for display has an event or input variable defined, use this subcommand instead of READ_FORMS.

- When you do not want any input from the terminal user, use this subcommand.

- Executing SHOW_FORMS:

  - Displays all the forms you have scheduled for display and have not deleted. If you added or combined forms since the last READ_FORMS or SHOW_FORMS subcommand, it displays them for the first time.

  - Removes from the screen the forms you deleted since the last READ_FORMS or SHOW_FORMS subcommand.

  - Displays variables replaced since the last READ_FORMS or SHOW_FORMS subcommand.

  - Displays objects with attributes set or reset since the last READ_FORMS or SHOW_FORMS subcommand.

MANAGE_FORMS Functions

You can use the following functions when managing forms with the MANAGE_FORMS utility.

$EVENT_NAME MANF Function

Purpose $EVENT_NAME returns the name of the event the application user executed to complete his or her interaction with a form.

Format **$EVENT_NAME**

Parameters None.

Remarks An event is usually executed when the user presses the return key or a function key.

MANAGE_FORMS Functions

You can use the following functions when managing forms with the MANAGE_FORMS utility.

$EVENT_NAME
MANF Function

Purpose $EVENT_NAME returns the name of the event the application user executed to complete his or her interaction with a form.

Format **$EVENT_NAME**

Parameters None.

Remarks An event is usually executed when the user presses the return key or a function key.

$EVENT_NORMAL MANF Function

Purpose

$EVENT_NORMAL returns a boolean value specifying whether the event the user executed is defined as normal. The type of each event is defined when the form is created. The value of $EVENT_NORMAL is TRUE when the event is a normal event; it is FALSE when the event is not a normal event.

Format

$EVENT_NORMAL

Parameters

None.

Remarks

- When an event is normal, variables are validated and updated.
- When an event is abnormal, variables are not validated or updated.

$EVENT_POSITION MANF Function

Purpose $EVENT_POSITION returns information about the position of the event executed by the application user to complete interaction with a form. The information returned is determined by the keyword you specify in the parameter.

Format **$EVENT_POSITION (OPTION: keyword)**

Parameters OPTION

The keyword that specifies the type of information to be returned about an event position. Use one of the following keywords:

CHARACTER_POSITION (CP)

Returns the character position within the object where the event occurred. The value returned is an integer; the first character position is 1. The CHARACTER_POSITION value is valid only if $EVENT_POSITION (OBJECT_EVENT) returns a TRUE value.

FORM_NAME (FN)

Returns the name of the form where the event occurred.

FORM_X_POSITION (FXP)

Returns the x position of the event on the form. The x position is an integer; 1 indicates the upper left corner of the form. The x position increases by 1 for each character, counting from left to right.

FORM_Y_POSITION (FYP)

Returns the y position of the event on the form. The y position is an integer; 1 indicates the upper left corner of the form. The y position increases by 1 for each character, counting from top to bottom.

OBJECT_EVENT (OE)

Returns a boolean value specifying whether the event occurred in an object on the form. The value returned is TRUE when the event occurred in an object and FALSE when it did not.

OBJECT_NAME (ON)

Returns the name of the object where the event occurred. The OBJECT_NAME value is valid only if $EVENT_POSITION (OBJECT_EVENT) returns a TRUE value.

OCCURRENCE (O)

Returns an integer indicating in which occurrence of the object the event occurred. The OCCURRENCE value is valid only if $EVENT_POSITION (OBJECT_EVENT) returns a TRUE value.

OBJECT_TYPE (OT)

Returns a keyword that indicates the type of object in which the event occurred. One of the following keywords is returned:

BOX
CONSTANT_TEXT
CONSTANT_TEXT_BOX
LINE
VARIABLE_TEXT
VARIABLE_TEXT_BOX

OBJECT_X_POSITION (OXP)

Returns the x position of the object on the form. The x position is an integer; 1 indicates the upper left corner of the form. The x position increases by 1 for each character, counting from left to right. The OBJECT_X_POSITION value is valid only if $EVENT_POSITION (OBJECT_EVENT) returns a TRUE value.

OBJECT_Y_POSITION (OYP)

Returns the y position of the object on the form. The y position is an integer; 1 indicates the upper left corner of the form. The y position increases by 1 for each character, counting from top to bottom. The OBJECT_Y_POSITION value is valid only if $EVENT_POSITION (OBJECT_EVENT) returns a TRUE value.

SCREEN_X_POSITION (SXP)

Returns the x position of the event on the screen. The x position is an integer; 1 indicates the upper left corner of the screen. The x position increases by 1 for each character, counting from left to right.

SCREEN_Y_POSITION (SYP)

Returns the y position of the event on the screen. The y position is an integer; 1 indicates the upper left corner of the screen. The y position increases by 1 for each character, counting from top to bottom.

Using COBOL to Manage Forms 3

Writing a Program to Use Forms 3-2
Copying Parameter Definitions 3-3
Copying Data Definitions 3-4
Calling Screen Formatting 3-5
Displaying and Removing Forms and Variable Data 3-5
Processing Events and Data 3-7
Processing Normal Events 3-7
Processing Abnormal Events 3-8
Running a Prototype of the Application 3-8
Example Program for Managing Forms with COBOL 3-10
Forms Managed in the Program 3-10
Design Specification 3-13
Form Definition Decks 3-15
Example COBOL Program 3-16

Expanding and Compiling a Program 3-30

Helping the User Start the Application 3-32
Creating a User Procedure 3-32
Creating a User Prolog 3-33
Selecting a Natural Language 3-34
Starting the Application 3-34

COBOL Subroutine Calls for Interacting with Forms 3-35
Adding a Form 3-36
Changing Table Size 3-38
Closing a Form 3-40
Combining Forms 3-41
Deleting a Form 3-43
Getting an Integer Variable 3-45
Getting the Next Event 3-48
Getting a Real Variable 3-52
Getting a Record 3-55
Getting a String Variable 3-58
Opening a Form 3-61
Popping a Form 3-63
Positioning a Form 3-64
Pushing a Form 3-66
Reading a Form 3-67
Replacing an Integer Variable 3-69
Replacing a Real Variable 3-72
Replacing a Record 3-75
Replacing a String Variable 3-77
Resetting a Form 3-79

Resetting an Object Attribute 3-80
Setting the Cursor Position 3-82
Setting Line Mode 3-84
Setting an Object Attribute 3-85
Showing Forms 3-87

Using COBOL to Manage Forms 3

Chapter 1 presented an example of creating and managing forms. It demonstrated that both the designer and the programmer have specific tasks to accomplish. When creating forms, and then managing the forms using a COBOL program, the following tasks need to be accomplished:

1. The form designer and programmer plan the forms and program.

2. The form designer creates the forms specifying COBOL as the form processor (or programming language) and prepares a design specification.

3. The form designer puts the forms in an object library and makes the form record definition available. Each record definition contains the data definitions of all variables defined on a particular form and is written in COBOL.

4. The programmer codes the program, including calls to Screen Formatting COBOL subroutines based on the design specification. These calls manage the forms created by the designer.

5. The programmer expands and compiles the program.

6. The programmer writes a user procedure to start the application and helps the user set up the correct terminal environment for using the forms.

When the last task is complete, the program and forms are ready for the application user.

This chapter describes the tasks performed by the programmer and shows them being executed in a COBOL program. At the end of the chapter you will find format and parameter descriptions for each call to COBOL subroutines used by Screen Formatting.

The designer's tasks and, also, the formats of the CYBIL procedure calls that create forms are described in chapter 7. (For information about designing forms using the Screen Design Facility, see the NOS/VE Screen Design Facility manual instead.)

Writing a Program to Use Forms

When writing a program to use forms, you must:

- Copy the parameter definitions provided by Screen Formatting.
- Copy the data definitions generated by Screen Formatting when the designer creates the form. The data definitions hold values transferred to and from the form for the variable text objects.
- Call Screen Formatting subroutines to manage the forms and the variable text objects on the forms.

To better understand how the forms and variables interrelate, you can also run a prototype of the application. This is usually done before you write the program. In the prototype, you interactively make calls to Screen Formatting to display each form. You then interact with the form.

Following are descriptions of the tasks you must accomplish in your COBOL program. After the descriptions is an explanation of how to run a prototype and an example COBOL program showing how the tasks are executed.

Copying Parameter Definitions

To obtain the values for the COBOL status parameter, copy the FDE$COBOL_STATUS deck into your program. The following example shows some of the contents of this deck:

```
01  FDE-COBOL-STATUS USAGE COMP PIC S9(18) SYNC LEFT.
    88 FDE-REQUEST-SUCCESSFUL VALUE 0.
    88 FDE-TERMINAL-DISCONNECTED VALUE 1.
    88 FDE-NO-INPUT-REQUEST VALUE 2.
    88 FDE-CURSOR-NOT-IN-VARIABLE VALUE 3.
```

To obtain the values for the COBOL variable status parameter, copy the FDE$COBOL_VARIABLE_STATUS deck into your program. The following example shows some of the contents of this deck:

```
01  FDE-COBOL-VARIABLE-STATUS USAGE COMP PIC S9(18) SYNC LEFT.
    88 FDE-NO-ERROR VALUE 0.
    88 FDE-INVALID-STRING VALUE 1.
    88 FDE-INVALID-REAL VALUE 2.
```

Appendix D has a complete list of the contents of both decks.

When checking the status of subroutines, you must check the FDE-COBOL-STATUS parameter and also check, when present, the FDE-COBOL-VARIABLE-STATUS parameter. If the value of FDE-COBOL-STATUS is zero, you can process output from the subroutine. However, if there is also a FDE-COBOL-VARIABLE-STATUS parameter, check its value before using variable output from the subroutine. The variable status is independent of the status for the subroutine.

Copying Data Definitions

The data definitions for each form reside on a *form definition record* created by the form designer. In your program, you transfer data to and from variable text objects through this record.

When the designer creates a form, Screen Formatting generates a common deck that defines the form definition record. For example, Screen Formatting[1] generated the following source file for a form named COBOL-SELECT-FORM. (The form definition record name is the same as the form name.)

```
*DECK COBOL_SELECT_FORM expand = false
    01 COBOL-SELECT-FORM.
        03 SELECT-MESSAGE PIC X(40).
        03 OBJECT PIC X(1).
```

The designer saves this file as a deck on a NOS/VE SOURCE_CODE_UTILITY (SCU) library.[2] The DECK directive in the file creates the correct name for the deck when it is processed.

In the beginning of your program, you must copy the form definition deck for each form created by the designer:

- Get the name of the deck from the design specification (the designer assigns the name while creating the form).
- Copy the deck by specifying its name on either the SCU *COPY directive or the COBOL COPY statement.

1. For this example, Screen Formatting was accessed through the Screen Design Facility.

2. Because each form has its own definition and the STATUS parameters use common decks, we recommend that you manage the source text using SCU. (For information on SCU, see the NOS/VE Source Code Management manual.)

Calling Screen Formatting

When writing a program that uses forms, you perform two basic tasks with Screen Formatting subroutines:

- Displaying and removing forms and variable data on the application user's screen.
- Processing events executed by the user.

Displaying and Removing Forms and Variable Data

To control the display of forms and variable data on the user's screen, you perform the following steps in the sequence given:

1. Open the form.

   When you open a form, Screen Formatting locates it and allocates resources for processing the Screen Formatting calls that use the form.

   You need open a form only once, no matter how many times you use or update it. For this reason, begin a procedure by opening all the forms you will use. When a form requires a large amount of storage for variables, however, you may want to open that one only when the application user needs it.

   (For the format of the call that opens forms, see *Opening a Form* later in this chapter).

2. Add the form.

   When you add a form, Screen Formatting schedules it for display on the application user's screen.

   To display more than one form at a time, add all the forms before you display them. Screen Formatting maintains a list of all forms you add. The last form you add to the list becomes the top form on the screen. Because forms are opaque, the top form covers other forms appearing in the same area.

   When the terminal user completes data entry, the cursor position indicates what form Screen Formatting should process. Variables on this form (and any forms combined with this one) are validated and updated. Variables on other forms are not updated or validated.

   (For the formats of the calls that schedule forms for display, see *Adding a Form* and *Combining Forms* later in this chapter.)

3. Read the form.

   When you read a form, Screen Formatting displays all the forms you've added.

   When a form has an event or input variable defined, reading forms also accepts data from the application user and displays values returned by the program.

   (For the format of the call that reads forms, see *Reading Forms* later in this chapter. When none of the forms scheduled for display has an event or input variable defined, you can use a similar call described in *Showing Forms* later in this chapter.)

4. Delete the form.

   When you delete a form, Screen Formatting deletes it from the list of forms scheduled for display. The next time you read forms, the deleted form is removed from the screen. However, the form remains available for later use in the program (you must reschedule it for display).

   (For the format of the call that deletes a form, see *Deleting a Form* later in this chapter.)

5. Close the form.

   When you close a form, Screen Formatting releases the resources the form uses. The form is no longer available to the user or your program.

   (For the format of the call that closes a form, see *Closing a Form* later in this chapter.)

Processing Events and Data

When creating a form, the designer defines two types of events a user can execute to return control to the program: normal and abnormal.

- For normal events, the program performs requested actions such as getting variables, doing computations, and updating the form.
- For abnormal events, the program takes its own action. You generally then delete the form and go on, or stop the program.

Processing Normal Events

To process a normal event:

1. Get the name of the event and the position of the cursor from Screen Formatting.

   Screen Formatting validates the data the user enters (the form designer defined the validation rules) and transfers values of screen variables to its storage. The form designer may also have created error forms to be displayed when the user enters an incorrect value or presses a key not defined as an event.

   (For the format of the call that gets the event name and cursor position, see *Getting the Next Event* at the end of this chapter.)

2. Get the data from Screen Formatting storage and transfer it to program storage.

   (For formats of the calls that get data, see the following sections later in this chapter: *Getting a Record, Getting an Integer Variable, Getting a Real Variable,* and *Getting a String Variable.*)

3. Replace the data in Screen Formatting storage with the data in program storage.

   (For formats of the calls that replace variables, see the following sections later in this chapter: *Replacing a Record, Replacing an Integer Variable, Replacing a Real Variable,* and *Replacing a String Variable.*)

You can also reset the variables on a form to their original state. (For formats of the calls that reset variables to their original state, see *Resetting a Form* and *Resetting an Object Attribute* later in this chapter.)

Processing Abnormal Events

To process an abnormal event:

1. Get the name of the event and the position of the cursor from Screen Formatting.

   Unlike a normal event, Screen Formatting neither validates user entries nor transfers values of screen variables to Screen Formatting storage.

   (For the format of the call that gets the event name and cursor position, see *Getting the Next Event* later in this chapter.)

2. Write your own procedure to perform the task the design specification assigns to the event. Typical actions for an abnormal event include:

   - Resetting a form and redisplaying it.
   - Moving the user to a new form for additional processing.
   - Returning the user to a previous form.
   - Stopping the program.

   The user's screen is updated when you either read the forms again or end the program.

Running a Prototype of the Application

Once the forms have been created for your application, you can interactively run a prototype using the MANAGE_FORMS utility. This allows you to test the order in which the forms appear, and to interact with the forms as the application user will do.

An example of an application prototype is given in chapter 2 under the section named The Application Prototype. This prototype uses forms that were created specifically for use in an SCL procedure. To learn about using a prototype, you can run the one described in the Prototype section.

Once you are familiar with the utility, you can also run a prototype using forms created for a COBOL program. However, because the prototype uses a NOS/VE utility, the variables created for the forms must conform to the SCL naming conventions. Often the variables created for COBOL use the hyphen. This character is not allowed in SCL. Therefore, when Screen Formatting creates the variables for the prototype, it automatically converts the hyphen to an underscore.

To use the prototype with the COBOL forms on the library specified in the prototype example rather than opening the SCL forms listed, open the forms named:

COBOL_SELECT_FORM
COBOL_RECTANGLE_FORM
COBOL_CIRCLE_FORM

One variable of type RECORD is created for each form as described for the SCL forms shown in the prototype example. The variable has the same name as the form. You can display the data structure of each variable using the DISPLAY_VALUE command. For example, to display the data structure for the COBOL_RECTANGLE_FORM variable, enter the following command:

```
mf/display_value value=cobol_rectangle_form ..
mf../display_options=data_structure
display option: DATA_STRUCTURE

"RECORD"
  SIDE_TABLE: "ARRAY"
               1. "RECORD"
                   SIDE: "INTEGER"  0
                  "RECORD END"

               2. "RECORD"
                   SIDE: "INTEGER"  0
                  "RECORD END"
             "ARRAY END"

RECTANGLE_AREA: "INTEGER"  0
RECTANGLE_MESSAGE: "STRING"  '
"RECORD END"
```

For more information about the structure of variables that are records, see the NOS/VE System Usage manual.

Example Program for Managing Forms with COBOL

The program in this example computes the area of circles and rectangles. The example includes:

- Pictures of the forms managed in the program.
- The design specification supplied by the form designer.
- The form definition decks.
- The example program.

Forms Managed in the Program

The example program manages three forms residing on an object library named EXAMPLE_OBJECT_LIBRARY that must be in the user's command list.

When a user starts the application, Select Form appears (figure 3-1).

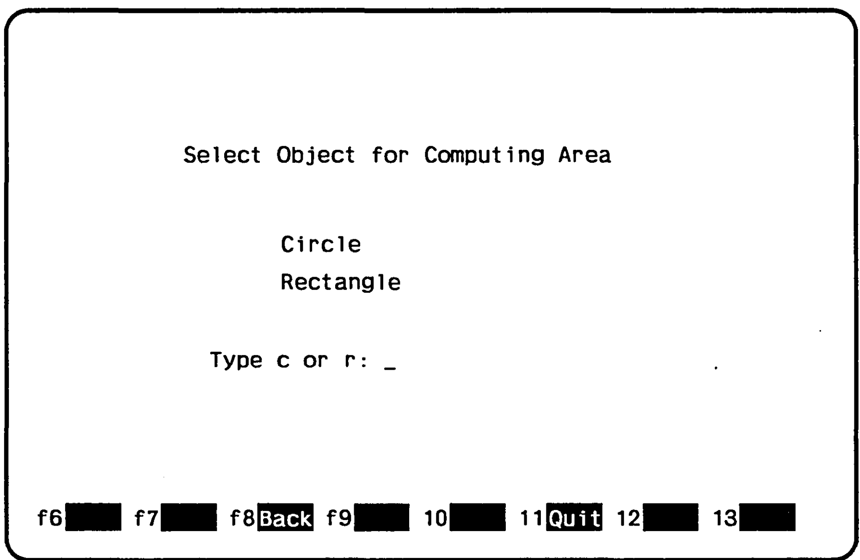


Figure 3-1. Select Form

On Select Form, a user enters either *c* to compute the area of a circle or *r* to compute the area of a rectangle.

When a user enters *r* on Select Form, Rectangle Form (figure 3-2) appears.

```
Compute Area of Rectangle

                              Type height: ________
  Area is:

  Type width: ________

f6     f7     f8 Back  f9     10     11 Quit  12     13
```

Figure 3-2. Rectangle Form

On Rectangle Form, the user enters the lengths of the sides of the rectangle as integers and presses the return key to have the program compute the area.

When a user enters *c* on Select Form, Circle Form (figure 3-3) appears.

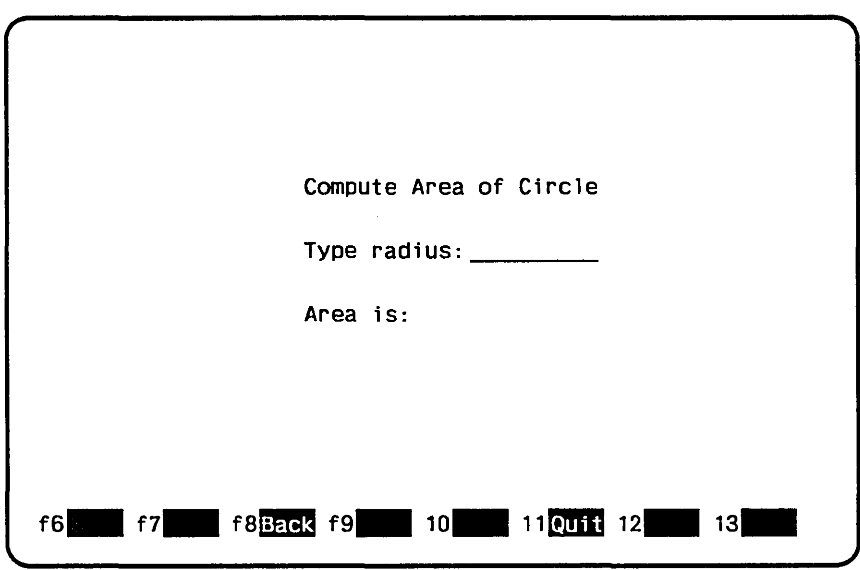


Figure 3-3. Circle Form

On Circle Form, the user enters the radius of the circle as a real value and presses the return key to have the program compute the area.

Design Specification

In writing the example program, the programmer uses the information the form designer listed in the following design specification:

- The names for the three forms used by the program are:

  COBOL_SELECT_FORM
  COBOL_RECTANGLE_FORM
  COBOL_CIRCLE_FORM

- The user can call both the Rectangle Form and Circle Form from the Select Form.

- The following variable text objects are defined on the forms:

Variable Object	Description
Select Form:	
OBJECT	Area for user input of *r* or *c*.
SELECT_MESSAGE	Area for displaying error messages.
Rectangle Form:	
TABLE	Table that holds values for the rectangle's sides.
SIDE	Areas (two) for user input of values for the rectangle's sides.
RECTANGLE_AREA	Area for returning value of computed area.
RECTANGLE_MESSAGE	Area for displaying error messages.
Circle Form:	
RADIUS	Area for user input of value for the circle's radius.
CIRCLE_AREA	Area for returning value of computed area.
CIRCLE_MESSAGE	Area for displaying error messages.

- The following events are defined on the forms:

Event	Description
COMPUTE	A normal program event that processes data the user entered on the form. For Select Form, the COMPUTE event checks whether the user entered *r* or *c* and then displays the appropriate form. For the other forms, COMPUTE calculates the area and redisplays the form.
BACK	An abnormal program event that takes the user back to a previous environment. For Select Form, the BACK event stops the program. For the other forms, BACK returns the user to Select Form.
QUIT	An abnormal program event that stops the program.

Form Definition Decks

When the designer creates the three forms (by writing a program or using Screen Design Facility), a form definition record is created with each form. For the example program, the programmer copies the following form definition decks placed by the designer on an SCU library. The library in this example is named EXAMPLE_SOURCE_LIBRARY.

The COBOL_SELECT_FORM deck:

```
01  COBOL-SELECT-FORM.
    03  SELECT-MESSAGE PIC X(40).
    03  OBJECT PIC X(1).
```

The COBOL_RECTANGLE_FORM deck:

```
01  COBOL-RECTANGLE-FORM.
    03  SIDE-TABLE OCCURS 2.
        05  SIDE PIC S9(18)
           COMP SYNC LEFT.
    03  RECTANGLE-AREA PIC S9(18) COMP SYNC LEFT.
    03  RECTANGLE-MESSAGE PIC X(40).
```

The COBOL_CIRCLE_FORM deck:

```
01  COBOL-CIRCLE-FORM.
    03  CIRCLE-AREA COMP-1.
    03  RADIUS COMP-1.
    03  CIRCLE-MESSAGE PIC X(40).
```

Example COBOL Program

This COBOL program calls the forms and executes the events described in the previous sections. The program is in the SCU deck named COBOL_COMPUTE_OBJECT_AREA. To run the example program, see the Screen Formatting examples in the Examples online manual.

```
        IDENTIFICATION DIVISION.
        PROGRAM-ID. COMPUTEAREA.
        DATA DIVISION.
        WORKING-STORAGE SECTION.

       * Copy definitions for Screen Formatting conditions.

*COPY FDE$COBOL_STATUS
*COPY FDE$COBOL_VARIABLE_STATUS

       * Copy record for select form.

*COPY cobol_select_form

       * Copy record for circle form.

*COPY cobol_circle_form

       *   Copy record for rectangle form.

*COPY cobol_rectangle_form

        01  CHARACTER-POSITION
              USAGE COMP PIC S9(18) SYNC LEFT.
        01  CIRCLE-FORM-IDENTIFIER
              USAGE COMP PIC S9(18) SYNC LEFT.
        01  DISPLAY-NAME VALUE IS "ERROR" PIC X(31).
        01  EVENT-NAME PIC X(31).
        01  EVENT-NORMAL PIC X.
        01  EVENT-OBJECT-NAME PIC X(31).
        01  EVENT-OCCURRENCE USAGE COMP PIC S9(18) SYNC LEFT.
        01  EVENT-POSITION USAGE COMP PIC S9(18) SYNC LEFT.
        01  EVENT-TYPE USAGE COMP PIC S9(18) SYNC LEFT.
        01  FORM-IDENTIFIER USAGE COMP PIC S9(18) SYNC LEFT.
        01  FORM-NAME PICTURE X(31).
        01  FORM-X-POSITION USAGE COMP PIC S9(18) SYNC LEFT.
        01  FORM-Y-POSITION USAGE COMP PIC S9(18) SYNC LEFT.
        01  LAST-EVENT PIC X.
```

```
01  OCCURRENCE USAGE COMP PIC S9(18) SYNC LEFT.
01  OBJECT-TYPE USAGE COMP PIC S9(18) SYNC LEFT.
01  OBJECT-X-POSITION USAGE COMP PIC S9(18) SYNC LEFT.
01  OBJECT-Y-POSITION USAGE COMP PIC S9(18) SYNC LEFT.
01  PI COMP-1 VALUE 3.14.
01  RECTANGLE-FORM-IDENTIFIER
      USAGE COMP PIC S9(18) SYNC LEFT.
01  SCREEN-X-POSITION USAGE COMP PIC S9(18) SYNC LEFT.
01  SCREEN-Y-POSITION USAGE COMP PIC S9(18) SYNC LEFT.
01  SELECT-FORM-IDENTIFIER
      USAGE COMP PIC S9(18) SYNC LEFT.
01  VARIABLE-NAME PIC X(31).
01  VARIABLE-STATUS USAGE COMP PIC S9(18) SYNC LEFT.

PROCEDURE DIVISION.
BEGIN.

* Open all forms used by the program
* and assign form identifiers.

    MOVE "COBOL_SELECT_FORM" TO FORM-NAME.
    CALL "FDP$XOPEN_FORM" USING FORM-NAME
      SELECT-FORM-IDENTIFIER FDE-COBOL-STATUS.
    IF NOT FDE-REQUEST-SUCCESSFUL
      DISPLAY "Open failed on form select."
      STOP RUN
    END-IF.

    MOVE "COBOL_CIRCLE_FORM" TO FORM-NAME.
    CALL "FDP$XOPEN_FORM" USING FORM-NAME
      CIRCLE-FORM-IDENTIFIER FDE-COBOL-STATUS.
    IF NOT FDE-REQUEST-SUCCESSFUL
      DISPLAY "Open failed on form circle."
      STOP RUN
    END-IF.

    MOVE "COBOL_RECTANGLE_FORM" TO FORM-NAME.
    CALL "FDP$XOPEN_FORM" USING FORM-NAME
      RECTANGLE-FORM-IDENTIFIER FDE-COBOL-STATUS.
    IF NOT FDE-REQUEST-SUCCESSFUL
      DISPLAY "Open failed on form rectangle."
      STOP RUN
    END-IF.
```

```
* Add select form to list scheduled for display.

     CALL "FDP$XADD_FORM" USING SELECT-FORM-IDENTIFIER
       FDE-COBOL-STATUS.
     IF NOT FDE-REQUEST-SUCCESSFUL
       DISPLAY "Add failed on form select."
       STOP RUN
     END-IF.

* Update screen and accept user terminal entry
* for object; display all added forms.

 GET-OBJECT-INPUT.
     CALL "FDP$XREAD_FORMS" USING FDE-COBOL-STATUS.
     IF NOT FDE-REQUEST-SUCCESSFUL
       DISPLAY "Read failed on form select."
       STOP RUN
     END-IF.

* Get screen event that determine next actions.

     CALL "FDP$XGET_NEXT_EVENT" USING EVENT-NAME
       EVENT-NORMAL SCREEN-X-POSITION SCREEN-Y-POSITION
       FORM-IDENTIFIER FORM-X-POSITION FORM-Y-POSITION
       EVENT-TYPE EVENT-OBJECT-NAME EVENT-OCCURRENCE
       EVENT-POSITION OBJECT-TYPE OBJECT-X-POSITION
       OBJECT-Y-POSITION
       LAST-EVENT FDE-COBOL-STATUS.
     IF NOT FDE-REQUEST-SUCCESSFUL
       DISPLAY "Get event failed on form select."
       STOP RUN
     END-IF.

* Stop program on QUIT or BACK event.

     IF EVENT-NAME NOT EQUAL TO "COMPUTE"
       PERFORM STOP-PROGRAM
     END-IF.
```

```
* Transfer object variable from form to program.

     MOVE "OBJECT" TO VARIABLE-NAME.
     MOVE 1 TO OCCURRENCE.
     CALL "FDP$XGET_STRING_VARIABLE" USING
       SELECT-FORM-IDENTIFIER VARIABLE-NAME OCCURRENCE
       OBJECT FDE-COBOL-VARIABLE-STATUS FDE-COBOL-STATUS.
     IF NOT FDE-REQUEST-SUCCESSFUL
       DISPLAY "Get string failed on form select."
       STOP RUN
     END-IF.

* If terminal user entered invalid data, display
* error message and ask for another entry.

     IF NOT FDE-NO-ERROR THEN
      MOVE "Type r or c" TO SELECT-MESSAGE
      MOVE "SELECT-MESSAGE" TO VARIABLE-NAME
      CALL "FDP$XREPLACE_STRING_VARIABLE" USING
       SELECT-FORM-IDENTIFIER VARIABLE-NAME
       OCCURRENCE SELECT-MESSAGE
       FDE-COBOL-VARIABLE-STATUS FDE-COBOL-STATUS
      GO TO GET-OBJECT-INPUT
     END-IF.

     IF OBJECT EQUALS "R" THEN
* Remove select form and compute area of rectangle.

       CALL "FDP$XDELETE_FORM" USING
         SELECT-FORM-IDENTIFIER FDE-COBOL-STATUS
         IF NOT FDE-REQUEST-SUCCESSFUL
           DISPLAY "Delete failed on form select."
           STOP RUN
         END-IF
       PERFORM COMPUTE-RECTANGLE-AREA THRU CRA-END
     ELSE
       IF OBJECT EQUALS "C" THEN
```

```
* Remove select form and compute area of circle.

          CALL "FDP$XDELETE_FORM" USING
            SELECT-FORM-IDENTIFIER FDE-COBOL-STATUS
            IF NOT FDE-REQUEST-SUCCESSFUL
              DISPLAY "Delete failed on form select."
              STOP RUN
            END-IF
          PERFORM COMPUTE-CIRCLE-AREA THRU CCA-END
        ELSE

* Terminal user entered invalid value for object.
* Display error message and ask for another entry.

          MOVE "Type r or c." TO SELECT-MESSAGE
          MOVE "SELECT-MESSAGE" TO VARIABLE-NAME
          CALL "FDP$XREPLACE_STRING_VARIABLE" USING
            SELECT-FORM-IDENTIFIER VARIABLE-NAME
            OCCURRENCE SELECT-MESSAGE
            FDE-COBOL-VARIABLE-STATUS FDE-COBOL-STATUS
            IF NOT FDE-REQUEST-SUCCESSFUL
              DISPLAY
                "Replace string failed on form select."
              STOP RUN
            END-IF
          GO TO GET-OBJECT-INPUT
        END-IF
      END-IF.

* Process event from rectangle form or circle form.

      IF EVENT-NAME EQUALS "QUIT"
        PERFORM STOP-PROGRAM
      END-IF.


* A BACK event occurred; display select form in
* original state.

      CALL "FDP$XRESET_FORM" USING SELECT-FORM-IDENTIFIER
        FDE-COBOL-STATUS.
      IF NOT FDE-REQUEST-SUCCESSFUL
        DISPLAY "Reset failed on form select."
        STOP RUN
      END-IF.
```

```
    CALL "FDP$XADD_FORM" USING SELECT-FORM-IDENTIFIER
      FDE-COBOL-STATUS.
    IF NOT FDE-REQUEST-SUCCESSFUL
      DISPLAY "Add failed on form select."
      STOP RUN
    END-IF.

    GO TO GET-OBJECT-INPUT.

COMPUTE-CIRCLE-AREA.

* Display circle form in original state.

    CALL "FDP$XRESET_FORM" USING CIRCLE-FORM-IDENTIFIER
      FDE-COBOL-STATUS.
    IF NOT FDE-REQUEST-SUCCESSFUL
      DISPLAY "Reset failed on form circle."
      STOP RUN
    END-IF.

    CALL "FDP$XADD_FORM" USING CIRCLE-FORM-IDENTIFIER
      FDE-COBOL-STATUS.
      IF NOT FDE-REQUEST-SUCCESSFUL
        DISPLAY "Add failed on form circle."
        STOP RUN
      END-IF.

* Update screen and get radius from
* terminal user entry.

GET-CIRCLE-INPUT.
    CALL "FDP$XREAD_FORMS" USING FDE-COBOL-STATUS.
    IF NOT FDE-REQUEST-SUCCESSFUL
      DISPLAY "Read failed on form circle."
      STOP RUN
    END-IF.
```

```
           CALL "FDP$XGET_NEXT_EVENT" USING EVENT-NAME
             EVENT-NORMAL SCREEN-X-POSITION SCREEN-Y-POSITION
             FORM-IDENTIFIER FORM-X-POSITION FORM-Y-POSITION
             EVENT-TYPE EVENT-OBJECT-NAME EVENT-OCCURRENCE
             EVENT-POSITION OBJECT-TYPE OBJECT-X-POSITION
             OBJECT-Y-POSITION
             LAST-EVENT FDE-COBOL-STATUS.
           IF NOT FDE-REQUEST-SUCCESSFUL
             DISPLAY "Get event failed on form circle."
             STOP RUN
           END-IF.

           IF EVENT-NAME NOT EQUAL TO "COMPUTE"
             CALL "FDP$XDELETE_FORM" USING
               CIRCLE-FORM-IDENTIFIER FDE-COBOL-STATUS
               IF NOT FDE-REQUEST-SUCCESSFUL
                 DISPLAY "Delete failed on form circle."
                 STOP RUN
               END-IF
              GO TO CCA-END
           END-IF.

      * Transfer terminal user entry for radius to program.

           MOVE "RADIUS" TO VARIABLE-NAME.
           MOVE 1 TO OCCURRENCE.
           CALL "FDP$XGET_REAL_VARIABLE" USING
             CIRCLE-FORM-IDENTIFIER VARIABLE-NAME OCCURRENCE
             RADIUS FDE-COBOL-VARIABLE-STATUS FDE-COBOL-STATUS.
           IF NOT FDE-REQUEST-SUCCESSFUL
             DISPLAY "Get real failed on form circle."
             STOP RUN
           END-IF.

           IF NOT FDE-NO-ERROR THEN
             MOVE "Type valid value for radius." TO
               CIRCLE-MESSAGE
             MOVE "CIRCLE-MESSAGE" TO VARIABLE-NAME
             CALL "FDP$XREPLACE_STRING_VARIABLE" USING
               CIRCLE-FORM-IDENTIFIER VARIABLE-NAME
               OCCURRENCE CIRCLE-MESSAGE
               FDE-COBOL-VARIABLE-STATUS FDE-COBOL-STATUS
             GO TO GET-CIRCLE-INPUT
           END-IF.
```

```
* Compute area of circle and display it.

     COMPUTE CIRCLE-AREA = PI * RADIUS ** 2.

     MOVE "CIRCLE-AREA" TO VARIABLE-NAME.
     CALL "FDP$XREPLACE_REAL_VARIABLE" USING
       CIRCLE-FORM-IDENTIFIER VARIABLE-NAME OCCURRENCE
       CIRCLE-AREA FDE-COBOL-VARIABLE-STATUS
       FDE-COBOL-STATUS.
     IF NOT FDE-REQUEST-SUCCESSFUL
       DISPLAY
         "Replace real failed on form rectangle."
       STOP RUN
     END-IF.

     IF NOT FDE-NO-ERROR THEN

* Area value could not be displayed using
* output format defined for form.
* Revise the form or the program to accommodate
* the size of the number.

       MOVE "Format cannot display area." TO
           CIRCLE-MESSAGE
       MOVE "CIRCLE-MESSAGE" TO VARIABLE-NAME
       CALL "FDP$XREPLACE_STRING_VARIABLE" USING
         CIRCLE-FORM-IDENTIFIER VARIABLE-NAME
         OCCURRENCE CIRCLE-MESSAGE
         FDE-COBOL-VARIABLE-STATUS FDE-COBOL-STATUS
       GO TO GET-CIRCLE-INPUT
     END-IF.
```

```
* Blank error message in case previously displayed.

      MOVE SPACES TO CIRCLE-MESSAGE.
      MOVE "CIRCLE-MESSAGE" TO VARIABLE-NAME.
      CALL "FDP$XREPLACE_STRING_VARIABLE" USING
        CIRCLE-FORM-IDENTIFIER VARIABLE-NAME
        OCCURRENCE CIRCLE-MESSAGE
        FDE-COBOL-VARIABLE-STATUS FDE-COBOL-STATUS.
      IF NOT FDE-REQUEST-SUCCESSFUL
        DISPLAY "Replace string failed on form circle."
        STOP RUN
      END-IF.

* Process next user entry.

      GO TO GET-CIRCLE-INPUT.
  CCA-END.  EXIT.

  COMPUTE-RECTANGLE-AREA.

* Display rectangle form in original state.

      CALL "FDP$XRESET_FORM" USING
        RECTANGLE-FORM-IDENTIFIER FDE-COBOL-STATUS.
      IF NOT FDE-REQUEST-SUCCESSFUL
        DISPLAY "Reset failed on form rectangle."
        STOP RUN
      END-IF.

      CALL "FDP$XADD_FORM" USING
        RECTANGLE-FORM-IDENTIFIER FDE-COBOL-STATUS.
      IF NOT FDE-REQUEST-SUCCESSFUL
        DISPLAY "Add failed on form rectangle."
        STOP RUN
      END-IF.
```

```
* Update screen and get terminal user entry for
* rectangle height and width.

 GET-RECTANGLE-INPUT.
     CALL "FDP$XREAD_FORMS" USING FDE-COBOL-STATUS.
     IF NOT FDE-REQUEST-SUCCESSFUL
       DISPLAY "Read failed on form rectangle."
       STOP RUN
     END-IF.

     CALL "FDP$XGET_NEXT_EVENT" USING EVENT-NAME
       EVENT-NORMAL SCREEN-X-POSITION SCREEN-Y-POSITION
       FORM-IDENTIFIER FORM-X-POSITION FORM-Y-POSITION
       EVENT-TYPE EVENT-OBJECT-NAME EVENT-OCCURRENCE
       EVENT-POSITION OBJECT-TYPE OBJECT-X-POSITION
       OBJECT-Y-POSITION
       LAST-EVENT FDE-COBOL-STATUS.
     IF NOT FDE-REQUEST-SUCCESSFUL
       DISPLAY "Get event failed on form rectangle."
       STOP RUN
     END-IF.

* If abnormal event (BACK or QUIT) occurs,
* return to caller.

     IF EVENT-NAME NOT EQUAL TO "COMPUTE"
       CALL "FDP$XDELETE_FORM" USING
         RECTANGLE-FORM-IDENTIFIER FDE-COBOL-STATUS
         IF NOT FDE-REQUEST-SUCCESSFUL
           DISPLAY "Delete failed on form rectangle."
           STOP RUN
         END-IF
        GO TO CRA-END
     END-IF.
```

```
* Remove any previous error indications.

     MOVE SPACES TO RECTANGLE-MESSAGE.
     MOVE "RECTANGLE-MESSAGE" TO VARIABLE-NAME.
     CALL "FDP$XREPLACE_STRING_VARIABLE" USING
       RECTANGLE-FORM-IDENTIFIER VARIABLE-NAME
       OCCURRENCE RECTANGLE-MESSAGE
       FDE-COBOL-VARIABLE-STATUS FDE-COBOL-STATUS.
     IF NOT FDE-REQUEST-SUCCESSFUL
       DISPLAY
         "Replace string failed on form rectangle."
       STOP RUN
     END-IF.

     MOVE 1 TO OCCURRENCE.
     MOVE "SIDE" TO VARIABLE-NAME.
     CALL "FDP$XRESET_OBJECT_ATTRIBUTE" USING
       RECTANGLE-FORM-IDENTIFIER VARIABLE-NAME
       OCCURRENCE FDE-COBOL-STATUS
     MOVE 2 TO OCCURRENCE.
     CALL "FDP$XRESET_OBJECT_ATTRIBUTE" USING
       RECTANGLE-FORM-IDENTIFIER VARIABLE-NAME
       OCCURRENCE FDE-COBOL-STATUS

* Transfer height value from form to program.

     MOVE "SIDE" TO VARIABLE-NAME.
     MOVE 1 TO OCCURRENCE.
     CALL "FDP$XGET_INTEGER_VARIABLE" USING
       RECTANGLE-FORM-IDENTIFIER VARIABLE-NAME
       OCCURRENCE SIDE (1) FDE-COBOL-VARIABLE-STATUS
       FDE-COBOL-STATUS.
     IF NOT FDE-REQUEST-SUCCESSFUL
       DISPLAY "Get integer failed on form rectangle."
       STOP RUN
     END-IF.
```

```
* If data invalid, move cursor to height value
* and display error message.

     IF NOT FDE-NO-ERROR THEN
       MOVE 1 TO CHARACTER-POSITION
       CALL "FDP$XSET_CURSOR_POSITION" USING
         RECTANGLE-FORM-IDENTIFIER VARIABLE-NAME
         OCCURRENCE CHARACTER-POSITION FDE-COBOL-STATUS
       CALL "FDP$XSET_OBJECT_ATTRIBUTE" USING
         RECTANGLE-FORM-IDENTIFIER VARIABLE-NAME
          OCCURRENCE DISPLAY-NAME FDE-COBOL-STATUS
       MOVE "Type valid value for height." TO
          RECTANGLE-MESSAGE
       MOVE "RECTANGLE-MESSAGE" TO VARIABLE-NAME
       CALL "FDP$XREPLACE_STRING_VARIABLE" USING
         RECTANGLE-FORM-IDENTIFIER VARIABLE-NAME
         OCCURRENCE RECTANGLE-MESSAGE
         FDE-COBOL-VARIABLE-STATUS FDE-COBOL-STATUS
       GO TO GET-RECTANGLE-INPUT
     END-IF.

* Transfer width value from form to program.

     MOVE 2 TO OCCURRENCE.
     CALL "FDP$XGET_INTEGER_VARIABLE" USING
       RECTANGLE-FORM-IDENTIFIER VARIABLE-NAME
       OCCURRENCE SIDE (2) FDE-COBOL-VARIABLE-STATUS
       FDE-COBOL-STATUS.
     IF NOT FDE-REQUEST-SUCCESSFUL
       DISPLAY "Get integer failed on form rectangle."
       STOP RUN
     END-IF.

* If data invalid, move cursor to width value and display
* error message.

     IF NOT FDE-NO-ERROR THEN
       MOVE 1 TO CHARACTER-POSITION
       CALL "FDP$XSET_CURSOR_POSITION" USING
         RECTANGLE-FORM-IDENTIFIER VARIABLE-NAME
         OCCURRENCE CHARACTER-POSITION FDE-COBOL-STATUS
        CALL "FDP$XSET_OBJECT_ATTRIBUTE" USING
         RECTANGLE-FORM-IDENTIFIER VARIABLE-NAME
         OCCURRENCE DISPLAY-NAME FDE-COBOL-STATUS
```

```
          MOVE "Type valid value for width."
            TO RECTANGLE-MESSAGE
          MOVE "RECTANGLE-MESSAGE" TO VARIABLE-NAME
          MOVE 1 TO OCCURRENCE
          CALL "FDP$XREPLACE_STRING_VARIABLE" USING
            RECTANGLE-FORM-IDENTIFIER VARIABLE-NAME
            OCCURRENCE RECTANGLE-MESSAGE
            FDE-COBOL-VARIABLE-STATUS FDE-COBOL-STATUS
          GO TO GET-RECTANGLE-INPUT
        END-IF.

* Compute area of rectangle and display it.

        MULTIPLY SIDE (1) BY SIDE (2) GIVING
          RECTANGLE-AREA.
        MOVE "RECTANGLE-AREA" TO VARIABLE-NAME.
        MOVE 1 TO OCCURRENCE.
        CALL "FDP$XREPLACE_INTEGER_VARIABLE" USING
            RECTANGLE-FORM-IDENTIFIER VARIABLE-NAME
            OCCURRENCE RECTANGLE-AREA
            FDE-COBOL-VARIABLE-STATUS FDE-COBOL-STATUS.
        IF NOT FDE-REQUEST-SUCCESSFUL
          DISPLAY
            "Replace integer failed on form rectangle."
          STOP RUN
        END-IF.

        IF NOT FDE-NO-ERROR THEN

* Area value could not be displayed using
* output format defined for form.
* Revise the form or the program to accommodate
* the size of the number.

          MOVE "Format cannot display area."
            TO RECTANGLE-MESSAGE
          MOVE "RECTANGLE-MESSAGE" TO VARIABLE-NAME
          MOVE 1 TO OCCURRENCE
          CALL "FDP$XREPLACE_STRING_VARIABLE" USING
            RECTANGLE-FORM-IDENTIFIER VARIABLE-NAME
            OCCURRENCE RECTANGLE-MESSAGE
            FDE-COBOL-VARIABLE-STATUS FDE-COBOL-STATUS
         GO TO GET-RECTANGLE-INPUT
        END-IF.
```

```
 * Process next user entry.

     GO TO GET-RECTANGLE-INPUT.
 CRA-END.  EXIT.

 STOP-PROGRAM.

* Close all forms and delete from list scheduled
* for display.

     CALL "FDP$XCLOSE_FORM" USING
       SELECT-FORM-IDENTIFIER FDE-COBOL-STATUS.
     IF NOT FDE-REQUEST-SUCCESSFUL
       DISPLAY "Close failed on form select."
     END-IF.

     CALL "FDP$XCLOSE_FORM" USING
       CIRCLE-FORM-IDENTIFIER FDE-COBOL-STATUS.
     IF NOT FDE-REQUEST-SUCCESSFUL
       DISPLAY "Close failed on form circle."
     END-IF.

     CALL "FDP$XCLOSE_FORM" USING
       RECTANGLE-FORM-IDENTIFIER FDE-COBOL-STATUS.
     IF NOT FDE-REQUEST-SUCCESSFUL
       DISPLAY "Close failed on form rectangle."
     END-IF.

     STOP RUN.
```

Expanding and Compiling a Program

Programs using Screen Formatting use common decks and form definition records that reside outside the main program. To manage the source text for this type of program, put the program in one or more SCU decks. This allows you to update individual parts of a program and to use forms in more than one program without duplicating code.[3]

To expand and compile a program maintained in SCU decks:

1. Expand the deck containing the main program.

2. Compile the expanded program.

3. Put the compiled program on an object library.

A procedure for compiling and expanding a program is shown in the following example. (The example is based on the example program and form definition records described earlier. The example shows how to place decks on library EXAMPLE_SOURCE_LIBRARY.)

The procedure calls SCU to expand the SCU directives contained in the program. For this expansion, you must include the $SYSTEM.CYBIL.OSF$PROGRAM_INTERFACE library as an alternate base. The program is then compiled and put on an object library.

```
PROCEDURE cobol_compile_deck, cobcd (
  deck,  d: name = $required
  status)
  source_code_utility
    use_library base=example_source_library result=$null
    expand_deck deck=deck ..
    compile=$local.compile ..
    alternate_base=$system.cybil.osf$program_interface
  quit

  cobol input=$local.compile ..
    list=$local.listing runtime_checks=all  ..
     binary_object=$local.lgo  ..
    debug_aids=all
```

3. For information on SCU, see the NOS/VE Source Code Management manual.

```
    create_object_library
      add_module library=example_object_library
      combine_module library=$local.lgo
      generate_library library=example_object_library.$next
    quit

  PROCEND cobol_compile_deck
```

To use the procedure, put it on library EXAMPLE_OBJECT_LIBRARY and then add the library to your command list (using the CREATE_COMMAND_LIST_ENTRY command). You can execute the procedure by entering:

```
/cobol_compile_deck deck=cobol_compute_object_area
```

The compiled program is now also on library EXAMPLE_OBJECT_LIBRARY.

For more information on writing and using procedures, see the NOS/VE System Usage manual.

Helping the User Start the Application

The complete application consists of your program and the forms created by the designer. To integrate the forms with your program, you must:

- Create a procedure that gives users access to the object library containing the forms and program.
- Ensure that the user's terminal environment is set up properly to use the forms (in most instances, by creating a user prolog).
- Ensure that users select the correct natural language.
- Ensure that users know how to start the application.

Creating a User Procedure

To give the user access to the object library containing the forms:

1. Write a NOS/VE procedure from which the user starts the application.
2. Place the procedure on the library that contains the compiled program.

For example, the following procedure executes the application that uses the starting procedure COMPUTEAREA on library EXAMPLE_OBJECT_LIBRARY. The other libraries accessed by the program are $SYSTEM.FDF$LIBRARY and $SYSTEM.TDU.TERMINAL_DEFINITIONS. Users must have these libraries available in order for the program to call the Screen Formatting subroutines.

```
PROCEDURE cobol_compute_area, cobca (
  status)

  execute_task ..
    library=(example_object_library,$system.fdf$library,..
    $system.tdu.terminal_definitions) ..
    starting_procedure=computearea

PROCEND cobol_compute_area
```

Creating a User Prolog

To ensure that the users' terminal environment is set up properly to use the forms, make sure they set the following terminal characteristics before they execute the procedure:

Characteristic	Description
Terminal model	Identifies the terminal to NOS/VE.
Attention character	Provides a character users can enter to interrupt the application.
Hold messages	Tells the network to hold all network messages until the user stops the application. Otherwise, a computer operator message may overwrite a form while a user is entering data, confusing the user.

In most instances, users should set up their terminal for the entire terminal session in their user prologs. The example below does the following:

- Identifies a Digital Equipment Corporation VT220 terminal to the system.
- Chooses the exclamation point as a way to interrupt the program.
- Holds all messages from a NAMVE/CDCNET network.
- Sets up the way the terminal uses the exclamation point to interrupt the program.

The users add the following commands to their user prologs:

```
change_terminal_attributes terminal_model=dec_vt220 ..
  attention_character='!' ..
  status_action=hold
change_term_conn_defaults attention_character_action=1
change_connection_attributes terminal_file_name=input aca=1
change_connection_attributes terminal_file_name=output aca=1
change_connection_attributes terminal_file_name=command aca=1
```

For a further explanation of how to interrupt a screen application during an interactive session, and what commands to use for networks other than NAMVE/CDCNET, see the NOS/VE System Usage manual.

Selecting a Natural Language

To ensure that users receive messages in the correct natural language, have them add the CHANGE_NATURAL_LANGUAGE command to their prologs. Because the default language is US_ENGLISH and all messages returned by Screen Formatting are in this language, have users include this command only when you have changed messages to another language.

Changing messages to other languages is described in the NOS/VE Object Code Management manual. The CHANGE_NATURAL_LANGUAGE command is described in the NOS/VE System Usage manual.

Starting the Application

To start the application, the users enter:

```
/create_command_list_entry e=example_object_library
/cobol_compute_area
```

When finished with the application, the users remove the object library from their command lists:

```
/delete_command_list_entry e=example_object_library
```

COBOL Subroutine Calls for Interacting with Forms

The subroutines that follow are used by Screen Formatting to manage forms. These subroutines are external routines that reside on the library called $SYSTEM.FDF$LIBRARY. To execute your program, users must have this library in their program library lists.

For each subroutine, there is a purpose description, input format, list of parameters and their types, condition identifiers, and pertinent remarks.

When checking the status of subroutines, you must check the FDE-COBOL-STATUS parameter and also check, when present, the FDE-COBOL-VARIABLE-STATUS parameter. If the value of FDE-COBOL-STATUS is zero, you can process output from the subroutine. However, if there is also a FDE-COBOL-VARIABLE-STATUS parameter, check its value before using variable output from the subroutine. The variable status is independent of the status for the subroutine.

Adding a Form

Purpose

FDP$XADD_FORM schedules a form for display on the application user's screen.

Format

CALL "FDP$XADD_FORM" USING form-identifier fde-cobol-status

Parameters

form-identifier {input}

The identifier established when the form was opened. Include the following data description entry:

```
01 form-identifier
   USAGE COMP PIC S9(18) SYNC LEFT.
```

fde-cobol-status {output}

The variable that indicates the results of the subroutine. This variable is defined with the SCU *COPY FDE$COBOL_VARIABLE_STATUS directive you put in the program.

Conditions

The following conditions apply to this call and are defined as COBOL condition names in appendix D.

fde-bad-data-value
fde-form-already-added
fde-form-pushed
fde-form-too-large-for-screen
fde-invalid-form-identifier
fde-no-space-available
fde-system-error

Remarks

- When you call either the FDP$XREAD_FORMS or FDP$XSHOW_FORMS subroutine, Screen Formatting displays the added form on the terminal screen. The added form is placed on top of other forms occupying the same area on the screen.
- When displayed, each form that is added operates independently from other forms that have been added. When a user executes a normal event, Screen Formatting validates and updates only those variables on the form associated with the event. To have forms share events, see Combining Forms later in this section.
- Before you add a form, you must open it.
- You cannot add a pushed form.

Changing Table Size

Purpose FDP$XCHANGE_TABLE_SIZE changes the size of the table during program execution.

Format **CALL "FDP$XCHANGE_TABLE_SIZE" USING form-identifier table-name table-size fde-cobol-status**

Parameters form-identifier {input}

The identifier established when the form was opened. Include the following data description entry:

```
01 form-identifier
   USAGE COMP PIC S9(18) SYNC LEFT.
```

table-name {input}

The name of the table to change in size. Include the following data description entry:

```
01 table_name PIC X(31).
```

table-size {input}

The size of the table. While this subroutine is in effect, Screen Formatting limits the number of stored occurrences allowed for a table to the value you specify on this parameter. How many occurrences are displayed at one time depends on the number of visible occurrences defined in the form.

If you specify zero for the table size, no occurrences appear on the form.

Include the following data description entry:

```
01 table-size
   USAGE COMP PIC S9(18) SYNC LEFT.
```

fde-cobol-status {output}

The variable that indicates the results of the subroutine. This variable is defined with the SCU *COPY FDE$COBOL_VARIABLE_STATUS directive you put in the program.

Conditions

The following conditions apply to this call and are defined as COBOL condition names in appendix D.

fde-bad-data-value
fde-form-pushed
fde-invalid-form-identifier
fde-invalid-table-name
fde-invalid-table-size
fde-no-space-available
fde-unknown-table-name

Remarks

- The table must be present in an open form.
- The size limitation remains in effect until the next time you call the FDP$XCHANGE_TABLE_SIZE subroutine.
- The maximum size for a table is identified by the form as the maximum number of stored occurrences. If you specify a table size larger than the maximum, you receive an error message (fde-invalid-table-size).

Examples

The following examples describe how changing the size of a table affects the application user. On the form, the table's specifications are a maximum of 20 stored occurrences, of which 6 occurrences can be visible at one time.

- If you specify a table size of 10, Screen Formatting displays 6 occurrences and allows the application user to page to the 10th occurrence.
- If you specify a table size of 4, Screen Formatting displays 4 occurrences and does not allow the application user to page.

Closing a Form

Purpose FDP$XCLOSE_FORM releases resources used to process a form and deletes the form from the list scheduled for display.

Format **CALL "FDP$XCLOSE_FORM" USING form-identifier fde-cobol-status**

Parameters form-identifier {input}

The identifier established when the form was opened. Include the following data description entry:

```
01 form-identifier
   USAGE COMP PIC S9(18) SYNC LEFT.
```

fde-cobol-status {output}

The variable that indicates the results of the subroutine. This variable is defined with the SCU *COPY FDE$COBOL_STATUS directive you put in the program.

Conditions The following conditions apply to this call and are defined as COBOL condition names in appendix D.

fde-bad-data-value
fde-invalid-form-identifier
fde-form-pushed
fde-no-space-available

Remarks

- When the program calls either the FDP$XREAD_FORMS or FDP$XSHOW_FORMS subroutine, Screen Formatting removes the closed form from the terminal screen as a result of calling this procedure.
- Before you can close a form, you must open it.
- You cannot close a pushed form.

Combining Forms

Purpose

FDP$XCOMBINE_FORM combines a form with a previously added form and schedules the combined form for display on the terminal screen.

Format

CALL "FDP$XCOMBINE_FORM" USING added-form-identifier combine-form-identifier fde-cobol-status

Parameters

added-form-identifier {input}

The identifier for this instance of the previously added form. Include the following data description entry:

```
01 added-form-identifier
   USAGE COMP PIC S9(18) SYNC LEFT.
```

combine-form-identifier {input}

The identifier for the form you are combining with the previously added form. Include the following data description entry:

```
01 combine-form-identifier
   USAGE COMP PIC S9(18) SYNC LEFT.
```

fde-cobol-status {output}

The variable that indicates the results of the subroutine. This variable is defined with the SCU *COPY FDE$COBOL_VARIABLE_STATUS directive you put in the program.

Conditions

The following conditions apply to this call and are defined as COBOL condition names in appendix D.

fde-bad-data-value
fde-form-already-added
fde-form-already-combined
fde-form-pushed
fde-form-too-large-for-screen
fde-invalid-form-identifier
fde-no-space-available
fde-system-error

Remarks

- You cannot combine a pushed form.
- The combined form inherits the event definitions of the previously added form.
- Before you combine a form with a previously added form, you must open both forms.
- When the program calls either the FDP$XREAD_FORMS or FDP$XSHOW_FORMS subroutine, Screen Formatting displays the combined form. The combined form is placed on top of other forms occupying the same area on the screen.
- When the application user executes an event to return normally to the program, Screen Formatting updates all program variables associated with both the added and combined forms.
- To combine several forms with a previously added form, call this subroutine more than once.

Deleting a Form

Purpose

FDP$XDELETE_FORM deletes a form from the list of forms scheduled for display.

Format

CALL "FDP$XDELETE_FORM" USING form-identifier fde-cobol-status

Parameters

form-identifier {input}

The identifier established when the form was opened. Include the following data description entry:

```
01 form-identifier
   USAGE COMP PIC S9(18) SYNC LEFT.
```

fde-cobol-status {output}

The variable that indicates the results of the subroutine. This variable is defined with the SCU *COPY FDE$COBOL_VARIABLE_STATUS directive you put in the program.

Conditions

The following conditions apply to this call and are defined as COBOL condition names in appendix D.

fde-bad-data-value
fde-form-not-scheduled
fde-form-pushed
fde-invalid-form-identifier
fde-no-space-available

Remarks

- When the program calls either the FDP$XREAD_FORMS or FDP$XSHOW_FORMS subroutine, Screen Formatting removes the deleted form from the terminal screen and replots any forms uncovered by the deleted form with the next screen update.
- When you add a form (FDP$XADD_FORM) again that you previously deleted, the data in the form is retained.

- Before you delete a form, you must open it.
- You cannot delete a pushed form.
- If the form was added and has any combined forms associated with it, the combined forms are also deleted.
- When you delete a combined form, only that form is deleted. Areas covered by the combined form are replotted after the combined form is deleted.

Getting an Integer Variable

Purpose

FDP$XGET_INTEGER_VARIABLE gets the value the user entered on a form for an integer variable and transfers it to the program.

Format

CALL "FDP$XGET_INTEGER_VARIABLE" USING form-identifier name occurrence variable fde-cobol-variable-status fde-cobol-status

Parameters

form-identifier {input}

The identifier established when the form was opened. Include the following data description entry:

```
01 form-identifier
   USAGE COMP PIC S9(18) SYNC LEFT.
```

name {input}

The name of the integer variable to get and transfer to the program. The name was defined when the form was created.

occurrence {input}

The occurrence of the variable name. Include the following data description entry:

```
01 occurrence
   USAGE COMP PIC S9(18) SYNC LEFT.
```

variable {output}

The integer variable that Screen Formatting generates automatically in the form definition record. If you do not want to use the automatically generated variable, include the following data description entry:

```
01 variable
   USAGE COMP PIC S9(18) SYNC LEFT.
```

fde-cobol-variable-status {output}

The condition name that describes the status of the integer variable. The following values are possible:

FDE-INVALID-INTEGER

The user entered data that is not in the range defined for variable.

FDE-LOSS-OF-SIGNIFICANCE

The user entered an integer that is too large.

FDE-NO-ERROR

No error occurred.

This variable is defined with the SCU *COPY FDE$COBOL_VARIABLE_STATUS directive you put in the program.

fde-cobol-status {output}

The variable that indicates the subroutine results. This variable is defined with the SCU *COPY FDE$COBOL_STATUS directive you put in the program.

Conditions

The following conditions apply to this call and are defined as COBOL condition names in appendix D.

fde-bad-data-error
fde-invalid-form-identifier
fde-invalid-variable-name
fde-no-space-available
fde-system-error
fde-unknown-occurrence
fde-unknown-variable-name
fde-wrong-variable-type

Remarks

- Before you get an integer variable, you must open its form. If you get the variable after opening the form and before reading or replacing the variable on the form, the program returns the initial value specified by the form designer.

- If the form designer specifies data validation rules and error processing to display an error message or form, the program does not need to look at the variable status parameter.

  If the form designer specifies data validation rules and no error processing, the program must look at the variable status parameter.

  If the form designer specifies no data validation rules, the program must look at the variable status parameter.

Getting the Next Event

Purpose FDP$XGET_NEXT_EVENT gets the event resulting from the most recent FDP$XREAD_FORMS subroutine.

Format **CALL "FDP$XGET_NEXT_EVENT" USING event-name event-normal screen-x-position screen-y-position form-identifier form-x-position form-y-position event-type object-name object-occurrence character-position object-type object-x-position object-y-position last-event fde-cobol-status**

Parameters event-name {output}

A data name to receive the application user's event. Include the following data description entry:

```
01 event-name PIC X(31).
```

event-normal {output}

A data name to receive the event normal indication. If the event is normal, T is returned; if the event is not normal, F is returned. Include the following data description entry:

```
01 event-normal PIC X(1).
```

screen-x-position {output}

A data name to receive the x position of the event on the screen. The character position in the upper left corner of the screen is 1; the x position increases by 1 for each character on the screen counting from left to right. Include the following data description entry:

```
01 screen-x-position
   USAGE COMP PIC S9(18) SYNC LEFT.
```

screen-y-position {output}

A data name to receive the y position of the event on the screen. The character position in the upper left corner of the screen is 1; the y position increases by 1 for each character on the screen counting from top to bottom. Include the following data description entry:

```
01 screen-y-position
   USAGE COMP PIC S9(18) SYNC LEFT.
```

form-identifier {output}

The identifier established when the form was opened. Include the following data description entry:

```
01 form-identifier
   USAGE COMP PIC S9(18) SYNC LEFT.
```

form-x-position {output}

A data name to receive the x position of the event on the form. The character in the upper left corner of the form is 1; the x position increases by 1 for each character you count from left to right. Include the following data description entry:

```
01 form-x-position
   USAGE COMP PIC S9(18) SYNC LEFT.
```

form-y-position {output}

A data name to receive the y position of the event on the form. The character in the upper left corner of the form is 1; the y position increases by 1 for each character you count from top to bottom. Include the following data description entry:

```
01 form-y-position
   USAGE COMP PIC S9(18) SYNC LEFT.
```

event-type {output}

The event type. The following values are possible:

Value	Event Type
0	The event occurred on an area of a form containing no object.
1	The event occurred on a form object.

Include the following data description entry:

```
01 event-type
   USAGE COMP PIC S9(18) SYNC LEFT.
```

object-name {output}

When event-type is 1, the variable returns a value giving the name of the object on which the event occurred. Include the following data description entry:

```
01 object-name PIC X(31).
```

object-occurrence {output}

When event-type is 1, the variable returns a value giving the occurrence of the object name. Include the following data description entry:

```
01 object-occurrence
   USAGE COMP PIC S9(18) SYNC LEFT.
```

character-position {output}

When event-type is 1, the variable returns a value giving the character position within the object where the event occurred. The first character position is 1. Include the following data description entry:

```
01 character-position
   USAGE COMP PIC S9(18) SYNC LEFT.
```

object-type {output}

When event-type is 1, the variable indicates the type of object on which the event occurred. The following values are possible:

Value	Object Type
0	Box
1	Constant text
2	Constant text box
3	Line
5	Variable text
6	Variable text box

Include the following data description entry:

```
01 object-type
   USAGE COMP PIC S9(18) SYNC LEFT.
```

object-x-position {output}

When event-type is 1, the value returned is the x origin position of the object. The character in the upper left corner of the form is 1; the x position increases by 1 for each character you count from left to right. Include the following data description entry:

```
01 object-x-position
   USAGE COMP PIC S9(18) SYNC LEFT.
```

object-y-position {output}

When event-type is 1, the value returned is the y origin position of the object. The character in the upper left corner of the form is 1; the y position increases by 1 for each character you count from top to bottom. Include the following data description entry:

```
01 object-y-position
   USAGE COMP PIC S9(18) SYNC LEFT.
```

last-event {output}

Indicates whether this is the last event. The following values are possible:

Value	Meaning
T	This is the last event.
F	This is not the last event.

Include the following data description entry:

```
01 last-event PIC X(1).
```

fde-cobol-status {output}

The variable that indicates the results of the subroutine. This variable is defined with the SCU *COPY FDE$COBOL_STATUS directive you put in the program.

Conditions The following conditions apply to this call and are defined as COBOL condition names in appendix D.

fde-bad-data-value

Remarks The FDP$XREAD_FORMS subroutine deletes existing events. If the event is normal, Screen Formatting updates the variables in the added and combined forms containing the event. Later, you can request the transfer of these variables to program storage. If the event is abnormal, Screen Formatting does not update or validate variables.

Getting a Real Variable

Purpose

FDP$XGET_REAL_VARIABLE gets a value the user entered on a form for a real variable and transfers it to the program.

Format

CALL "FDP$XGET_REAL_VARIABLE" USING form-identifier name occurrence variable fde-cobol-variable-status fde-cobol-status

Parameters

form-identifier {input}

The identifier established when the form was opened. Include the following data description entry:

```
01 form-identifier
   USAGE COMP PIC S9(18) SYNC LEFT.
```

name {input}

The name of the variable to get. The name was defined when the form was created.

occurrence {input}

The occurrence of the variable name. Include the following data description entry:

```
01 occurrence
   USAGE COMP PIC S9(18) SYNC LEFT.
```

variable {output}

The value of the real variable that Screen Formatting generates automatically in the form definition record. If you do not want to use the automatically generated variable, include the following data description entry:

```
01 variable COMP-1.
```

fde-cobol-variable-status {output}

The condition that gives you the status of the variable. The following values are possible:

FDE-INDEFINITE

The user entered an indefinite number.

FDE-INVALID-BDP-DATA

The user entered data that does not correspond to the defined data type.

FDE-INVALID-REAL

The user entered data that is not within the range of real numbers defined for the variable.

FDE-LOSS-OF-SIGNIFICANCE

The user entered a number too large to be converted to the defined real program type.

FDE-NO-ERROR

No error occurred on the variable.

FDE-OVERFLOW

The user entered an exponent that is too large.

FDE-UNDERFLOW

The user entered an exponent that is too small.

This variable is defined with the SCU *COPY FDE$COBOL_VARIABLE_STATUS directive you put in the program.

fde-cobol-status {output}

The variable that indicates the results of the subroutine. This variable is defined with the SCU *COPY FDE$COBOL_STATUS directive you put in the program.

Conditions The following conditions apply to this call and are defined as COBOL condition names in appendix D.

fde-bad-data-value
fde-invalid-form-identifier
fde-invalid-variable-name
fde-no-space-available
fde-system-error
fde-unknown-occurrence
fde-unknown-variable-name

Remarks

- Before you get a real variable, you must open the form on which the user enters the value. If you get the variable after opening the form and before reading or replacing the variable on the form, the program returns the initial value specified by the form designer.

- If the form designer specifies data validation rules and error processing to display an error message or form, your program does not need to look at the variable status parameter.

  If the form designer specifies data validation rules and no error processing, the program must look at the variable status parameter.

  If the form designer specifies no data validation rules, the program must look at the variable status parameter.

Getting a Record

Purpose FDP$XGET_RECORD transfers the values of the form record to the program record.

Format **CALL "FDP$XGET_RECORD" USING form-identifier record fde-cobol-variable-status fde-cobol-status**

Parameters form-identifier {input}

The identifier established when the form was opened. Include the following data description entry:

```
01 form-identifier
   USAGE COMP PIC S9(18) SYNC LEFT.
```

record {output}

The name of the record that contains working storage information for the form. When the form is created, Screen Formatting generates the variable definition entries in this record. It is the program work area for the variables used on the form.

fde-cobol-variable-status {output}

The condition that gives you the status of the variable. The following values are possible:

FDE-INDEFINITE

The user entered an indefinite number.

FDE-INFINITE

The user entered an infinite number.

FDE-INVALID-BDP-DATA

The user entered data that does not correspond to the defined data type.

FDE-INVALID-INTEGER

The user entered data that is not within the range of integer numbers defined for the variable.

FDE-INVALID-REAL

The user entered data that is not within the range of real numbers defined for the variable.

FDE-INVALID-STRING

The user entered data that does not match the strings defined as valid for the variable.

FDE-LOSS-OF-SIGNIFICANCE

The user entered a number too large to be converted to the defined real or integer data type.

FDE-NO-DIGITS

The user, who should have entered a real or integer number, did not enter digits.

FDE-NO-ERROR

No error occurred on the variable.

FDE-OVERFLOW

The user entered an exponent that is too large.

FDE-UNDERFLOW

The user entered an exponent that is too small.

This variable is defined with the SCU *COPY FDE$COBOL_VARIABLE_STATUS directive you put in the program.

fde-cobol-status {output}

The variable that indicates the results of the subroutine. This variable is defined with the SCU *COPY FDE$COBOL_STATUS directive you put in the program.

Conditions The following conditions apply to this call and are defined as COBOL condition names in appendix D.

ide-bad-data-value
fde-form-has-no-variables
fde-invalid-form-identifier
fde-no-space-available
fde-system-error
fde-work-invalid

Remarks

- Before you get a record for a form, you must open the form. If you get the record after opening the form and before reading or replacing the record, the program returns the initial value specified by the form designer.

- If the form designer specifies data validation rules and error processing to display an error message or form, your program does not need to look at the variable status parameter.

  If the form designer specifies data validation rules and no error processing, the program must look at the variable status parameter.

  If the form designer specifies no data validation rules, the program must look at the variable status parameter.

Getting a String Variable

Purpose FDP$XGET_STRING_VARIABLE gets a value the user entered on a form for a string variable and transfers it to the program.

Format **CALL "FDP$XGET_STRING_VARIABLE" USING form-identifier name occurrence variable fde-cobol-variable-status fde-cobol-status**

Parameters form-identifier {input}

The identifier established when the form was opened. Include the following data description entry:

```
01 form-identifier
   USAGE COMP PIC S9(18) SYNC LEFT.
```

name {input}

The name of the variable to get. The name was defined when the form was created.

occurrence {input}

The occurrence of the variable name. Include the following data description entry:

```
01 occurrence
   USAGE COMP PIC S9(18) SYNC LEFT.
```

variable {output}

The variable that Screen Formatting generates automatically in the form definition record. The form definition record defines the variable. If you do not want to use the automatically generated variable, include the following data description entry (where n is the length of the variable):

```
01 variable PIC X(n).
```

fde-cobol-variable-status {output}

The condition that gives you the status of the variable. The following values are possible:

FDE-INVALID-STRING

The user entered a variable that does not match the strings defined for the variable.

FDE-NO-ERROR

No error occurred on the variable.

FDE-VARIABLE-TRUNCATED

The storage length of the parameter variable is not long enough.

This variable is defined with the SCU *COPY FDE$COBOL_VARIABLE_STATUS directive you put in the program.

fde-cobol-status {output}

The variable that indicates the results of subroutine. This variable is defined with the SCU *COPY FDE$COBOL_STATUS directive you put in the program.

Conditions The following conditions apply to this call and are defined as COBOL condition names in appendix D.

fde-bad-data-value
fde-invalid-form-identifier
fde-invalid-variable-name
fde-no-space-available
fde-system-error
fde-unknown-occurrence
fde-unknown-variable-name
fde-wrong-variable-name

Remarks

- Before you get a string variable, you must open the form on which the user enters the value. If you get the variable after opening the form and before reading or replacing the variable on the form, the program returns the initial value specified by the form designer.

- If the form designer specifies data validation rules and error processing to display an error message or form, your program does not need to look at the variable status parameter.

  If the form designer specifies data validation rules and no error processing, the program must look at the variable status parameter.

  If the form designer specifies no data validation rules, the program must look at the variable status parameter.

Opening a Form

Purpose FDP$XOPEN_FORM locates a form and and prepares it for use by the program.

Format **CALL "FDP$XOPEN_FORM" USING form-name form-identifier fde-cobol-status**

Parameters form-name {input}

The name of the form you want to open. Include the following data description entry:

```
01 form-name PIC X(31).
```

form-identifier {input-output}

The form identifier established for the form. Other Screen Formatting subroutines use this identifier when referencing the form. Include the following data description entry:

```
01 form-identifier
   USAGE COMP PIC S9(18) SYNC LEFT.
```

fde-cobol-status {output}

The variable that indicates the results of the subroutine. This variable is defined with the SCU *COPY FDE$COBOL_STATUS directive you put in the program.

Conditions The following conditions apply to this call and are defined as COBOL condition names in appendix D.

fde-bad-data-value
fde-form-already-open
fde-form-not-ended
fde-form-requires-conversion
fde-invalid-form-identifier
fde-invalid-form-name
fde-no-space-available
fde-system-error
fde-terminal-not-identified
fde-unknown-form-name

Remarks

- Screen Formatting locates a form as follows:
  - If the form name is blank, Screen Formatting assumes that the form identifier specifies the required dynamically created form.
  - If the form name is not blank, Screen Formatting searches the list of ended dynamically created forms.
  - If the form name is not blank and is not in the list of ended dynamically created forms, Screen Formatting searches the command library list to find the form name on the object code libraries. (You specify the order in which Screen Formatting searches the list using the NOS/VE command CREATE_COMMAND_LIST_ENTRY).
- Executing FDP$XOPEN_FORM does not display the form on the screen.
- The form identifier that FDP$XOPEN_FORM returns identifies the instance of open for a form. Forms dynamically created have only one instance of open. Forms stored on object libraries can have more than one instance of open. For each instance of open, Screen Formatting maintains the working environment (current value of variables and their display attributes) of the form.

Popping a Form

Purpose FDP$XPOP_FORMS deletes forms scheduled (added or combined) since the last FDP$XPUSH_FORMS call.

Format **CALL "FDP$XPOP_FORMS" USING fde-cobol-status**

Parameters fde-cobol-status {output}

The variable that indicates the results of the subroutine. This variable is defined with the SCU *COPY FDE$COBOL_STATUS directive you put in the program.

Conditions The following conditions apply to this call and are defined as COBOL condition names in appendix D.

fde-bad-data-value
fde-no-forms-to-pop

Remarks Events associated with the last list of pushed forms become active.

Positioning a Form

Purpose

FDP$XPOSITION_FORM schedules moving a form to a new location. Using this subroutine, you can define a form at one location and display it at another location, or you can move a form from where it is currently displayed to a new location.

Format

CALL "FDP$XPOSITION_FORM" USING form-identifier screen-x-position screen-y-position fde-cobol-status

Parameters

form-identifier {input}

The form identifier established when the form was opened. Include the following data description entry:

```
01 form-identifier
   USAGE COMP PIC S9(18) SYNC LEFT.
```

screen-x-position {input}

The x position on the screen for determining the upper left corner of the form. The character position in the upper left corner of the screen is 1, and the x position increases by 1 for each character on the screen counting from left to right. Include the following data description entry:

```
01 screen-x-position
   USAGE COMP PIC S9(18) SYNC LEFT.
```

screen-y-position {input}

The y position on the screen for determining the upper left corner of the form. The character position in the upper left corner of the screen is 1, and the y position increases by 1 for each character on the screen counting from top to bottom. Include the following data description entry:

```
01 screen-y-position
   USAGE COMP PIC S9(18) SYNC LEFT.
```

fde-cobol-status {output}

The variable that indicates the results of the subroutine. This variable is defined with the SCU *COPY FDE$COBOL_STATUS directive you put in the program.

Conditions

The following conditions apply to this call and are defined as COBOL condition names in appendix D.

fde-bad-data-value
fde-form-pushed
fde-form-too-large-for-screen
fde-invalid-form-identifier
fde-no-space-available
fde-system-error

Remarks

- When the program calls either the FDP$XREAD_FORMS or FDP$XSHOW_FORMS subroutine, Screen Formatting displays the form on the screen at the position specified in the call to FDP$XPOSITION_FORM.

- If you call this subroutine while the form is displayed, the form is deleted from its current location and added at the new location. The added form lays on top of any other form occupying the same area on the screen.

- If you call this procedure before the form is displayed, the form is displayed at the specified location.

- Before you position a form, you must open it.

- You cannot position a pushed form.

Pushing a Form

Purpose

FDP$XPUSH_FORMS causes Screen Formatting to record added and combined forms so you can return to them later.

Format

CALL "FDP$XPUSH_FORMS" USING fde-cobol-status

Parameters

fde-cobol-status {output}

The variable that indicates the results of the subroutine. This variable is defined with the SCU *COPY FDE$COBOL_STATUS directive you put in the program.

Conditions

The following conditions apply to this call and are defined as COBOL condition names in appendix D.

fde-bad-data-value
fde-no-forms-to-push

Remarks

- Events associated with these forms are not passed to the program.
- A program cannot change or close a pushed form.
- Pushed forms are displayed on the screen. If you want newly added forms to appear on a blank screen, first add a blank form that covers the screen.

  Updates to the screen continue to show the pushed forms.
- This subroutine deactivates the events associated with forms scheduled for display (added or combined) since the last push call.

Reading a Form

Purpose

FDP$XREAD_FORMS updates the terminal screen and accepts input from the application user.

Format

CALL "FDP$XREAD_FORMS" USING fde-cobol-status

Parameters

fde-cobol-status {output}

The variable that indicates the results of the subroutine. This variable is defined with the SCU *COPY FDE$COBOL_STATUS directive you put in the program.

Conditions

The following conditions apply to this call and are defined as COBOL condition names in appendix D.

fde-bad-data-value
fde-no-events-active
fde-no-forms-to-read
fde-system-error
fde-terminal-disconnected

Remarks

- A call to FDP$XREAD_FORMS:
  - Displays all the forms you scheduled for display and have not deleted. If you added or combined forms since the last FDP$XREAD_FORMS or FDP$XSHOW_FORMS call, it displays them for the first time.
  - Removes from the screen the forms you deleted since the last FDP$XREAD_FORMS or FDP$XSHOW_FORMS call.
  - Updates on the screen the variables replaced since the last FDP$XREAD_FORMS or FDP$XSHOW_FORMS call.
  - Updates on the screen the objects for which display attributes were set or reset since the last FDP$XREAD_FORMS or FDP$XSHOW_FORMS call.

- Events not retrieved with the FDP$XGET_NEXT_EVENT subroutine are deleted before any input is accepted from the user.
- The FDP$XREAD_FORMS subroutine does not execute unless the forms scheduled for display contain at least one active event.

Replacing an Integer Variable

Purpose

FDP$XREPLACE_INTEGER_VARIABLE transfers a program integer variable to Screen Formatting.

Format

CALL "FDP$XREPLACE_INTEGER_VARIABLE" USING form-indentifier name occurrence variable fde-cobol-variable-status fde-cobol-status

Parameters

form-identifier {input}

The identifier established when the form was opened. Include the following data description entry:

```
01 form-identifier
   USAGE COMP PIC S9(18) SYNC LEFT.
```

name {input}

The name of the integer variable to replace. The name was defined when the form was created.

```
01 name PIC X(31).
```

occurrence {input}

The occurrence of the variable name. Include the following data description entry:

```
01 occurrence
   USAGE COMP PIC S9(18) SYNC LEFT.
```

variable {input}

The integer variable that Screen Formatting generates automatically in the form definition record. If you do not want to use the automatically generated variable, include the following data description entry:

```
01 variable
   USAGE COMP PIC S9(18) SYNC LEFT.
```

fde-cobol-variable-status {output}

The condition that gives you the status of the variable. The following values are possible:

FDE-INVALID-INTEGER

The program supplied a value that is not within the range of integer numbers defined for the variable.

FDE-LOSS-OF-SIGNIFICANCE

The program supplied a value that is too large for the form variable.

FDE-NO-ERROR

No error occurred on the variable.

FDE-OUTPUT-FORMAT-BAD

The output format defined for the variable cannot output the variable.

This variable is defined with the SCU *COPY FDE$COBOL_VARIABLE_STATUS directive you put in the program.

fde-cobol-status {output}

The variable that indicates the results of the subroutine. This variable is defined with the SCU *COPY FDE$COBOL_STATUS directive you put in the program.

Conditions

The following conditions apply to this call and are defined as COBOL condition names in appendix D.

fde-bad-data-value
fde-form-pushed
fde-invalid-form-identifier
fde-invalid-variable-name
fde-no-space-available
fde-system-error
fde-unknown-occurrence
fde-unknown-variable-name
fde-wrong-variable-type

Remarks

- When you call either the FDP$XREAD_FORMS or FDP$XSHOW_FORMS subroutine, Screen Formatting replaces the integer variable on the terminal screen.
- Before you replace an integer variable, you must open the form on which it is replaced.
- You cannot replace an integer variable for a pushed form.
- If the integer variable is not valid, it is not replaced.

Replacing a Real Variable

Purpose

FDP$XREPLACE_REAL_VARIABLE transfers a program real variable to Screen Formatting.

Format

CALL "FDP$XREPLACE_REAL_VARIABLE" USING form-identifier name occurrence variable fde-cobol-variable-status fde-cobol-status

Parameters

form-identifier {input}

The identifier established when the form was opened. Include the following data description entry:

```
01 form-identifier
   USAGE COMP PIC S9(18) SYNC LEFT.
```

name {input}

The name of the real variable to replace. The name was defined when the form was created.

occurrence {input}

The occurrence of the variable name. Include the following data description entry:

```
01 occurrence
   USAGE COMP PIC S9(18) SYNC LEFT.
```

variable {input}

The value of the real variable that Screen Formatting generates automatically in the form definition record. If you do not want to use the automatically generated variable, include the following data description entry:

```
01 variable COMP-1.
```

fde-cobol-variable-status {output}

The condition that gives you the status of the variable. The following values are possible:

FDE-INVALID-REAL

The value the program supplied is not within the range of real numbers defined for the variable.

FDE-LOSS-OF-SIGNIFICANCE

The value the program supplied is too large for the form variable.

FDE-NO-ERROR

No error occurred on the variable.

FDE-OUTPUT-FORMAT-BAD

The output format defined for the variable cannot output the variable.

This variable is defined with the SCU *COPY FDE$COBOL_VARIABLE_STATUS directive you put in the program.

fde-cobol-status {output}

The variable that indicates the results of the subroutine. This variable is defined with the SCU *COPY FDE$COBOL_STATUS directive you put in the program.

Conditions

The following conditions apply to this call and are defined as COBOL condition names in appendix D.

fde-bad-data-value
fde-form-pushed
fde-invalid-form-identifier
fde-no-space-available
fde-system-error
fde-unknown-occurrence
fde-unknown-variable-name
fde-variable-name
fde-wrong-variable-type

Remarks

- When you call either the FDP$XREAD_FORMS or FDP$XSHOW_FORMS subroutine, Screen Formatting replaces the real variable on the terminal screen.
- Before you replace a real variable, you must open the form on which it is replaced.
- You cannot replace a real variable for a pushed form.
- If the real variable is not valid, it is not replaced.

Replacing a Record

Purpose

FDP$XREPLACE_RECORD transfers values of program variables to Screen Formatting for later display on a form.

Format

CALL "FDP$XREPLACE_RECORD" USING form-identifier record fde-cobol-variable-status fde-cobol-status

Parameters

form-identifier {input}

The identifier established when the form was opened. Include the following data description entry:

```
01 form-identifier
   USAGE COMP PIC S9(18) SYNC LEFT.
```

record {input}

The name of the record that contains working storage information for the form. When the form is created, Screen Formatting generates the variable definition entries in this record. It is the program work area for the variables used on the form.

fde-cobol-variable-status {output}

The condition that gives you the status of the variable. The following values are possible:

FDE-INDEFINITE

The program supplied an indefinite number.

FDE-INFINITE

The program supplied an infinite number.

FDE-LOSS-OF-SIGNIFICANCE

The program supplied a number too large to be converted to the form variable size.

FDE-NO-ERROR

No error occurred on the variable.

FDE-OUTPUT-FORMAT-BAD

The output format defined for the variable cannot output the variable.

FDE-OVERFLOW

The program supplied an exponent that is too large.

FDE-UNDERFLOW

The program supplied an exponent that is too small.

This variable is defined with the SCU *COPY FDE$COBOL_VARIABLE_STATUS directive you put in the program.

fde-cobol-status {output}

The variable that indicates the results of subroutine. This variable is defined with the SCU *COPY FDE$COBOL_STATUS directive you put in the program.

Conditions The following conditions apply to this call and are defined as COBOL condition names in appendix D.

fde-bad-data-value
fde-form-has-no-variables
fde-form-pushed
fde-invalid-form-identifier
fde-no-space-available
fde-work-invalid

Remarks

- When the program calls either the FDP$XREAD_FORMS or FDP$XSHOW_FORMS subroutine, Screen Formatting replaces the variables on the terminal screen with the values stored in Screen Formatting.
- Before you replace a record, you must open the form on which the variables are replaced.
- You cannot replace a record for a pushed form.

Replacing a String Variable

Purpose

FDP$XREPLACE_STRING_VARIABLE transfers a string variable to Screen Formatting.

Format

CALL "FDP$XREPLACE_STRING_VARIABLE" USING form-identifier name occurrence variable fde-cobol-variable-status fde-cobol-status

Parameters

form-identifier {input}

The identifier established when the form was opened. Include the following data description entry:

```
01 form-identifier
   USAGE COMP PIC S9(18) SYNC LEFT.
```

name {input}

The name of the string variable to replace. The name was defined when the form was created.

occurrence {input}

The occurrence of the variable name. Include the following data description entry:

```
01 occurrence
   USAGE COMP PIC S9(18) SYNC LEFT.
```

variable {input}

The variable that Screen Formatting generates automatically in the form definition record. If you do not want to use the automatically generated variable, include the following data description entry (n is the length of the variable):

```
01 variable PIC X(n).
```

fde-cobol-variable-status {output}

The condition that gives you the status of the variable. The following values are possible:

FDE-INVALID-STRING

The program supplied a variable that does not match the strings defined for the variable.

FDE-NO-ERROR

No error occurred on the variable.

This variable is defined with the SCU *COPY FDE$COBOL_VARIABLE_STATUS directive you put in the program.

fde-cobol-status {output}

The variable that indicates the results of the subroutine. This variable is defined with the SCU *COPY FDE$COBOL_STATUS directive you put in the program.

Conditions

The following conditions apply to this call and are defined as COBOL condition names in appendix D.

fde-bad-data-value
fde-form-pushed
fde-invalid-form-identifier
fde-invalid-variable-name
fde-no-space-available
fde-system-error
fde-unknown-occurrence
fde-unknown-variable-name
fde-wrong-variable-type

Remarks

- When the program calls either the FDP$XREAD_FORMS or FDP$XSHOW_FORMS subroutine, Screen Formatting replaces the string variable on the terminal screen.
- Before you replace a string variable, you must open the form on which it is replaced.
- You cannot replace a string variable for a pushed form.
- If the string variable is not valid, it is not replaced.
- If the form specifies that the data must be in uppercase, Screen Formatting converts it to uppercase before storing the data in the form.

Resetting a Form

Purpose

FDP$XRESET_FORM resets the form to the state specified by the form definition.

Format

CALL "FDP$XRESET_FORM" USING form-identifier fde-cobol-status

Parameters

form-identifier {input}

The identifier established when the form was opened. Include the following data description entry:

```
01 form-identifier
   USAGE COMP PIC S9(18) SYNC LEFT.
```

fde-cobol-status {output}

The variable that indicates the results of the subroutine. This variable is defined with the SCU *COPY FDE$COBOL_STATUS directive you put in the program.

Conditions

The following conditions apply to this call and are defined as COBOL condition names in appendix D.

fde-bad-data-value
fde-form-pushed
fde-invalid-form-identifier
fde-no-space-available
fde-system-error

Remarks

- When the program calls either the FDP$XREAD_ FORMS or FDP$XSHOW_FORMS subroutine, Screen Formatting displays the form on the terminal screen with the reset specifications.
- All variables belonging to the form have their initial values and display attributes. The form is in its defined position.
- Before you reset a form, you must open it.
- You cannot reset a pushed form.

Resetting an Object Attribute

Purpose

FDP$XRESET_OBJECT_ATTRIBUTE resets the display attributes for an object to those specified in the form definition.

Format

CALL "FDP$XRESET_OBJECT ATTRIBUTE" USING form-identifier object-name object-occurrence fde-cobol-status

Parameters

form-identifier {input}

The identifier established when the form was opened. Include the following data description entry:

```
01 form-identifier
   USAGE COMP PIC S9(18) SYNC LEFT.
```

object-name {input}

The name of the object whose attributes are reset. Include the following data description entry:

```
01 object-name PIC X(31).
```

object-occurrence {input}

The occurrence of the object. For the first or only occurrence, use 1. Include the following data description entry:

```
01 object-occurrence
   USAGE COMP PIC S9(18) SYNC LEFT.
```

fde-cobol-status {output}

The variable that indicates the results of the subroutine. This variable is defined with the SCU *COPY FDE$COBOL_STATUS directive you put in the program.

Conditions

The following conditions apply to this call and are defined as COBOL condition names in appendix D.

fde-bad-data-value
fde-form-not-scheduled
fde-form-pushed
fde-invalid-form-identifier
fde-invalid-object-name
fde-invalid-occurrence
fde-no-space-available
fde-unknown-object-name

Remarks

- You can reset the attributes of objects that are variable text, constant text, lines, or boxes.
- Before you reset the attribute of an object, you must open and either add or combine the form the object is on.
- When the program calls either the FDP$XREAD_FORMS or FDP$XSHOW_FORMS subroutine, Screen Formatting displays the object using the reset attributes.
- If the object you specify is not displayed on the screen, Screen Formatting shifts the data so the object is displayed (updates the screen automatically.)

Setting the Cursor Position

Purpose FDP$XSET_CURSOR_POSITION sets the cursor to a selected position for later display.

Format **CALL "FDP$XSET_CURSOR_POSITION" USING form-identifier object-name object-occurrence character-position fde-cobol-status**

Parameters form-identifier {input}

The identifier established when the form was opened. Include the following data description entry:

```
01 form-identifier
   USAGE COMP PIC S9(18) SYNC LEFT.
```

object-name {input}

The name of the object on which you want to set the cursor. Include the following data description entry:

```
01 object-name PIC X(31).
```

object-occurrence {input}

The integer specifying the occurrence of the object name. For the first occurrence, use 1. Include the following data description entry:

```
01 object-occurrence
   USAGE COMP PIC S9(18) SYNC LEFT.
```

character-position {input}

The character position to which you want to set the cursor. For the first character position, use 1. Include the following data description entry:

```
01 character-position
   USAGE COMP PIC S9(18) SYNC LEFT.
```

fde-cobol-status {output}

The variable that indicates the results of the subroutine. This variable is defined with the SCU *COPY FDE$COBOL_STATUS directive you put in the program.

Conditions

The following conditions apply to this call and are defined as COBOL condition names in appendix D.

fde-bad-data-value
fde-form-not-scheduled
fde-form-pushed
fde-invalid-character-position
fde-invalid-form-identifier
fde-invalid-object-name
fde-no-object-available-defined
fde-no-space-available
fde-system-error
fde-unknown-object-name
fde-unknown-occurrence

Remarks

- Use this subroutine to alter the default sequence of the application user's entry of variables. In the default sequence, Screen Formatting places the cursor on the first input variable of the highest priority form. The highest priority form is the form last added, combined, or positioned.

  At terminals with protected fields, the user tabs from one variable text object to the next. The cursor starts at the top line of the form; it moves from left to right on each line. When no variable text object appears on a line, the cursor moves down to the next line. At terminals without protected fields, the user must move the cursor using the arrow keys or the tab and return keys.

- When you call either the FDP$XREAD_FORMS or FDP$XSHOW_FORMS subroutine, Screen Formatting updates the terminal screen with the cursor at the specified position.

- If the position you specify is not visible on the screen, Screen Formatting shifts the data to make the cursor visible.

- The cursor position is in effect only for the next screen update from reading or showing forms.

- Before you set the cursor position on a form, you must open the form and either add or combine it.

- You cannot set the cursor position in a pushed form.

Setting Line Mode

Purpose FDP$XSET_LINE_MODE begins line-by-line interaction with an application user.

Format **CALL "FDP$XSET_LINE_MODE" USING fde-cobol-status**

Parameters fde-cobol-status {output}

The variable that indicates the results of the subroutine. This variable is defined with the SCU *COPY FDE$COBOL_STATUS directive you put in the program.

Conditions The following conditions apply to this call and are defined as COBOL condition names in appendix D.

fde-bad-data-value

Remarks

- Use this call for extended dialogues in line mode. For short dialogues, Screen Formatting automatically switches to the proper mode (line or screen) but resources used for screen mode interaction remain.
- This call releases all screen mode resources:
  - Open forms are closed.
  - The mode is set to line.

Setting an Object Attribute

Purpose

FDP$XSET_OBJECT_ATTRIBUTE changes a display attribute for an object.

Format

CALL "FDP$XSET_OBJECT_ATTRIBUTE" USING form-identifier object-name object-occurrence attribute-name fde-cobol-status

Parameters

form-identifier {input}

The identifier established when the form was opened. Include the following data description entry:

```
01 form-identifier
   USAGE COMP PIC S9(18) SYNC LEFT.
```

object-name {input}

The name of the object whose display attribute is being set. Include the following data description entry:

```
01 object-name PIC X(31).
```

object-occurrence {input}

The occurrence of the object. For the first or only occurrence, use 1. Include the following data description entry:

```
01 object-occurrence
   USAGE COMP PIC S9(18) SYNC LEFT.
```

attribute-name {input}

The name given to the display attribute when it was defined on the form. The attribute used here is defined for the form and not for a specific object. When using Screen Design Facility, screen attributes are defined through the ATTRIB function. When using a CYBIL program, the ADD_DISPLAY_DEFINITION attribute record defines form attributes.

Include the following data description entry:

```
01 attribute-name PIC X(31).
```

fde-cobol-status {output}

The variable that indicates the results of the subroutine. This variable is defined with the SCU *COPY FDE$COBOL_STATUS directive you put in the program.

Conditions The following conditions apply to this call and are defined as COBOL condition names in appendix D.

fde-bad-data-value
fde-form-not-scheduled
fde-form-pushed
fde-invalid-attribute-position
fde-invalid-form-identifier
fde-invalid-object-name
fde-invalid-occurrence
fde-no-space-available
fde-unknown-display-name
fde-unknown-object name
fde-unknown-occurrence

Remarks

- You can set the attributes of objects that are variable text, constant text, lines, or boxes.
- Changed attributes replace existing attributes.
- When you call either the FDP$XREAD_FORMS or FDP$XSHOW_FORMS subroutine, Screen Formatting displays the object using the set attributes.
- If the object you specify is not visible on the screen, Screen Formatting shifts the data to make the object visible.
- Before you set the attribute of an object, you must open the form the object is on and either add or combine it.
- You cannot set attributes of objects on a pushed form.

Showing Forms

Purpose FDP$XSHOW_FORMS updates the terminal screen.

Format **CALL "FDP$XSHOW_FORMS" USING fde-cobol-status**

Parameters fde-cobol-status {output}

The variable that indicates the results of the subroutine. This variable is defined with the SCU *COPY FDE$COBOL_STATUS directive you put in the program.

Conditions The following conditions apply to this call and are defined as COBOL condition names in appendix D.

fde-bad-data-value
fde-form-too-large-for-screen
fde-form-to-show
fde-no-space-available
fde-system-error
fde-terminal-disconnected

Remarks

- When none of the forms scheduled for display has an event or input variable defined, use this subroutine instead of FDP$XREAD_FORMS.

- When you do not want any input from the terminal user, use this subroutine.

- A call to FDP$XSHOW_FORMS:

  - Displays all the forms you scheduled for display and have not deleted. If you added or combined forms since the last FDP$XREAD_FORMS or FDP$XSHOW_FORMS call, it displays them for the first time.

  - Removes from the screen the forms you deleted since the last FDP$XREAD_FORMS or FDP$XSHOW_FORMS call.

- Displays variables replaced since last FDP$XREAD_FORMS or FDP$XSHOW_FORMS call.

- Displays objects with attributes set or reset since last FDP$XREAD_FORMS or FDP$XSHOW_FORMS call.

Using FORTRAN to Manage Forms 4

Writing a Program to Use Forms 4-2
Copying Data Definitions 4-3
Calling Screen Formatting 4-4
Displaying and Removing Forms and Variable Data 4-4
Processing Events and Data 4-6
Processing Normal Events 4-6
Processing Abnormal Events 4-7
Running a Prototype of the Application 4-7
Example Program for Managing Forms with FORTRAN 4-9
Forms Managed in the Program 4-9
Design Specification 4-12
Form Definition Decks 4-14
Example FORTRAN Program 4-15

Expanding and Compiling a Program 4-24

Helping the User Start the Application 4-26
Creating a User Procedure 4-26
Creating a User Prolog 4-27
Selecting a Natural Language 4-28
Starting the Application 4-28

FORTRAN Subroutine Calls for Interacting with Forms 4-29
Adding a Form 4-30
Changing Table Size 4-32
Closing a Form 4-34
Combining Forms 4-35
Deleting a Form 4-37
Getting an Integer Variable 4-39
Getting the Next Event 4-42
Getting a Real Variable 4-46
Getting a Record 4-49
Getting a String Variable 4-51
Opening a Form 4-54
Popping a Form 4-56
Positioning a Form 4-57
Pushing a Form 4-59
Reading Forms 4-60
Replacing an Integer Variable 4-61
Replacing a Real Variable 4-63
Replacing a Record 4-65
Replacing a String Variable 4-67
Resetting a Form 4-70
Resetting an Object Attribute 4-71

Setting the Cursor Position 4-73
Setting Line Mode 4-76
Setting an Object Attribute 4-77
Showing Forms 4-79

Using FORTRAN to Manage Forms 4

Chapter 1 presented an example of creating and managing forms. It demonstrated that both the designer and the programmer have specific tasks to accomplish. When creating forms, and then managing the forms using a FORTRAN program, the following tasks need to be accomplished:

1. The form designer and programmer plan the forms and program.

2. The form designer creates the forms specifying FORTRAN as the form processor (or programming language) and prepares a design specification.

3. The form designer puts the forms in an object library and makes the form record definition available. Each record definition contains the data definitions of all variables defined on a particular form and is written in FORTRAN.

4. The programmer codes the program, including calls to Screen Formatting FORTRAN subroutines based on the design specification. These calls manage the forms created by the designer.

5. The programmer expands and compiles the program.

6. The programmer writes a user procedure to start the application and helps the user set up the correct terminal environment for using the forms.

When the last task is complete, the program and forms are ready for the application user.

This chapter describes the tasks performed by the programmer and shows them being executed in a FORTRAN program. At the end of the chapter you will find format and parameter descriptions for each call to FORTRAN subroutines used by Screen Formatting.

The designer's tasks and, also, the formats of the CYBIL procedure calls that create forms are described in chapter 7. (For information about designing forms using the Screen Design Facility, see the NOS/VE Screen Design Facility manual instead.)

Writing a Program to Use Forms

When writing a program to use forms, you must:

- Copy the aliases used in FORTRAN for Screen Formatting subroutine. See appendix G, FORTRAN Call Definitions.
- Copy the data definitions generated by Screen Formatting when the designer creates the form. The data definitions hold values transferred to and from the form for the variable text objects.
- Call Screen Formatting subroutines to manage the forms and the variable text objects on the forms.

Following the descriptions of these tasks is a FORTRAN program in which these tasks are executed.

Copying Data Definitions

The data definitions for each form reside on a *form definition record* created by the form designer. In your program, you transfer data to and from variable text objects through this record.

When the designer creates a form, Screen Formatting generates a common deck that defines the form definition record. For example, Screen Formatting[1] generated the following source file for a form named SELECT. (The form definition record name is the same as the form name.)

```
*DECK SELECT expand = false
    CHARACTER SELECT*41
    CHARACTER XSELEC(41)
    EQUIVALENCE (SELECT,XSELEC(1))
    CHARACTER MESSAG*40
    EQUIVALENCE (XSELEC(1),MESSAG)
    CHARACTER OBJECT*1
    EQUIVALENCE (XSELEC(41),OBJECT)
```

The designer saves this file as a deck on a NOS/VE SOURCE_CODE_UTILITY (SCU) library.[2] The DECK directive in the file creates the correct name for the deck when it is processed.

Your program uses the names specified during form creation (select, messag, object). Names preceeded by the letter X are only used to assign names to FORTRAN storage locations.

In the beginning of your program, you must copy the form definition deck for each form created by the designer:

- Get the name of the deck from the design specification (the designer assigns the name while creating the form).

- Copy the deck by specifying its name on the SCU *COPY directive.

1. For this example, Screen Formatting was accessed through the Screen Design Facility.

2. Because each form has its own definition and the STATUS parameters use common decks, we recommend that you manage the source text using SCU. (For information on SCU, see the NOS/VE Source Code Management manual.)

Calling Screen Formatting

When you write a program that uses forms, you perform two basic tasks with Screen Formatting subroutines:

- Displaying and removing forms and variable data on the application user's screen.
- Processing events executed by the user.

Displaying and Removing Forms and Variable Data

To control the display of forms and variable data on the user's screen, you perform the following steps in the sequence given:

1. Open the form.

   When you open a form, Screen Formatting locates it and allocates resources for processing the Screen Formatting calls that use the form.

   You need open a form only once, no matter how many times you use or update it. For this reason, begin a procedure by opening all the forms you will use. When a form requires a large amount of storage for variables, however, you may want to open that one only when the application user needs it.

   (For the format of the call that opens forms, see *Opening a Form* later in this chapter).

2. Add the form.

   When you add a form, Screen Formatting schedules it for display on the application user's screen.

   To display more than one form at a time, add all the forms before you display them (the next step). The last form you schedule for display is the top form on the screen. Because forms are opaque, the top form covers other forms appearing in the same area. The cursor position indicates which form is ready for processing.

   (For the formats of the calls that schedule forms for display, see *Adding a Form* and *Combining Forms* later in this chapter.)

3. Read the form.

   When you read a form, Screen Formatting displays all the forms you've added.

   When a form has an event or input variable defined, reading forms also accepts data from the application user and displays values returned by the program.

   (For the format of the call that reads forms, see *Reading Forms* later in this chapter. When none of the forms scheduled for display has an event or input variable defined, you can use a similar call described in *Showing Forms* later in this chapter.)

4. Delete the form.

   When you delete a form, Screen Formatting deletes it from the list of forms scheduled for display. The next time you read forms, the deleted form is removed from the screen. However, the form remains available for later use in the program (you must reschedule it for display).

   (For the format of the call that deletes a form, see *Deleting a Form* later in this chapter.)

5. Close the form.

   When you close a form, Screen Formatting releases the resources the form uses. The form is no longer available to the user or your program.

   (For the format of the call that closes a form, see *Closing a Form* later in this chapter.)

Processing Events and Data

When creating a form, the designer defines two types of events a user can execute to return control to the program: normal and abnormal.

- For normal events, the program performs requested actions such as getting variables, doing computations, and updating the form.

- For abnormal events, the program takes its own action. You generally then delete the form and go on, or stop the program.

Processing Normal Events

To process a normal event:

1. Get the name of the event and the position of the cursor from Screen Formatting.

   Screen Formatting validates the data the user enters (the form designer defined the validation rules) and transfers values of screen variables to its storage. The form designer may also have created error forms to be displayed when the user enters an incorrect value or presses a key not defined as an event.

   (For the format of the call that gets the event name and cursor position, see *Getting the Next Event* at the end of this chapter.)

2. Get the data from Screen Formatting storage and transfer it to program storage.

   (For formats of the calls that get data, see the following sections later in this chapter: *Getting a Record, Getting an Integer Variable, Getting a Real Variable,* and *Getting a String Variable.*)

3. Replace the data in Screen Formatting storage with the data in program storage.

   (For formats of the calls that replace variables, see the following sections later in this chapter: *Replacing a Record, Replacing an Integer Variable, Replacing a Real Variable,* and *Replacing a String Variable.*)

You can also reset the variables on a form to their original state. (For formats of the calls that reset variables to their original state, see *Resetting a Form* and *Resetting an Object Attribute* later in this chapter.)

Processing Abnormal Events

To process an abnormal event:

1. Get the name of the event and the position of the cursor from Screen Formatting.

   Unlike a normal event, Screen Formatting neither validates user entries nor transfers values of screen variables to Screen Formatting storage.

   (For the format of the call that gets the event name and cursor position, see *Getting the Next Event* later in this chapter.)

2. Write your own procedure to perform the task the design specification assigns to the event. Typical actions for an abnormal event include:

   - Resetting a form and redisplaying it.
   - Moving the user to a new form for additional processing.
   - Returning the user to a previous form.
   - Stopping the program.

   The user's screen is updated when you either read the forms again or end the program.

Running a Prototype of the Application

Once the forms have been created for your application, you can interactively run a prototype using the MANAGE_FORMS utility. This allows you to test the order in which the forms appear, and to interact with the forms as the application user will do.

An example of an application prototype is given in chapter 2 under the section named The Application Prototype. This prototype uses forms that were created specifically for use in an SCL procedure. To learn about using a prototype, you can run the prototype as described in the section.

Once you are familiar with the utility, you can also run a prototype using forms created for a FORTRAN program. Because the naming conventions for FORTRAN do not conflict with SCL naming conventions, the variables defined for the form can be used in the prototype without any conversion taking place.

To use the prototype with the FORTRAN forms that are on the library specified in the prototype example rather than opening the SCL forms listed, open the forms named:

SELECT
RECTAN
CIRCLE

One variable of type RECORD is created for each form as described for the SCL forms shown in the prototype example. The variable has the same name as the form. You can display the data structure of each variable using the DISPLAY_VALUE command. For example, to display the data structure for the RECTAN variable, enter the following command:

```
mf/display_value value=rectan ..
mf../display_options=data_structure
display option: DATA_STRUCTURE

"RECORD"
  TABLE: "ARRAY"
           1. "RECORD"
               SIDE: "INTEGER"  0
              "RECORD END"

           2. "RECORD"
               SIDE: "INTEGER"  0
              "RECORD END"
         "ARRAY END"

  AREA: "INTEGER"  0
  MESSAG: "STRING"  '
"RECORD END"
```

For more information about the structure of variables that are records, see the NOS/VE System Usage manual.

Example Program for Managing Forms with FORTRAN

The program in this example computes the area of circles and rectangles. The example includes:

- Pictures of the forms managed in the program.
- The design specification supplied by the form designer.
- The form definition decks.
- The example program.

Forms Managed in the Program

The example program manages three forms residing on an object library named EXAMPLE_OBJECT_LIBRARY that must be in the user's command list.

When a user starts the application, Select Form appears (figure 4-1).

```
              Select Object for Computing Area

                        Circle
                        Rectangle

                   Type c or r: _


 f6      f7      f8Back  f9      10      11Quit  12      13
```

Figure 4-1. Select Form

On Select Form, a user enters either *c* to compute the area of a circle or *r* to compute the area of a rectangle.

When a user enters *r* on Select Form, Rectangle Form (figure 4-2) appears.

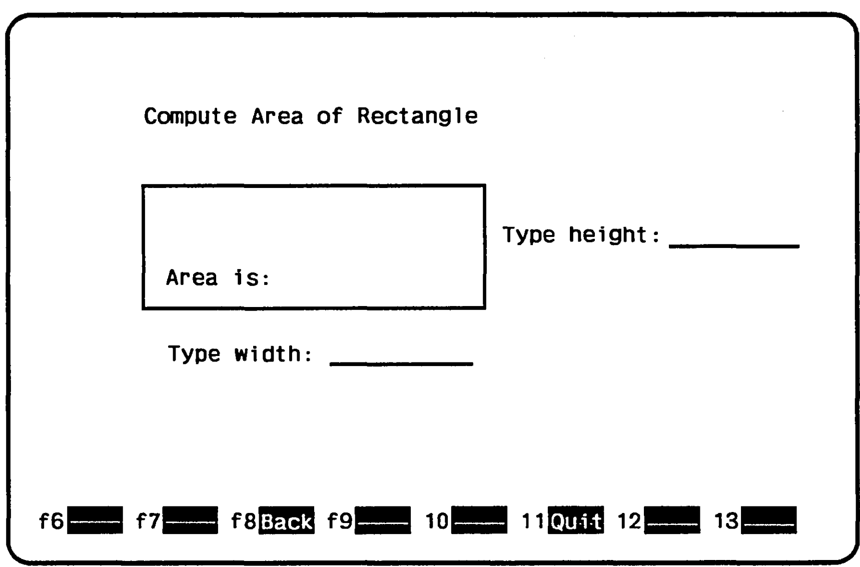


Figure 4-2. Rectangle Form

On Rectangle Form, the user enters the lengths of the sides of the rectangle as integers and presses the return key to have the program compute the area.

When a user enters *c* on Select Form, Circle Form (figure 4-3) appears.

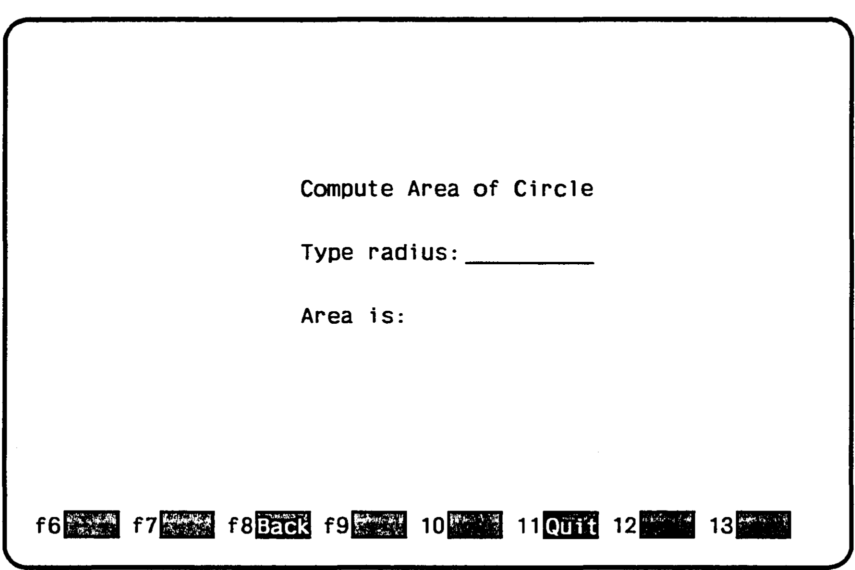


Figure 4-3. Circle Form

On Circle Form, the user enters the radius of the circle as a real value and presses the return key to have the program compute the area.

Design Specification

In writing the example program, the programmer uses the information the form designer listed in the following design specification:

- The names for the three forms used by the program are:

  SELECT (for Select Form)
  RECTAN (for Rectangle Form)
  CIRCLE (for Circle Form)

- The user can call both the Rectangle Form and Circle Form from the Select Form.

- The following variable text objects are defined on the forms:

Variable Object	Description
Select Form:	
MESSAG	Area for displaying error messages.
OBJECT	Area for user input of *r* or *c*.
Rectangle Form:	
SIDE	Areas (two) for user input of values for the rectangle's sides.
AREA	Area for returning value of computed area.
MESSAG	Area for displaying error messages.
Circle Form:	
RADIUS	Area for user input of value for the circle's radius.
AREA	Area for returning value of computed area.
MESSAG	Area for displaying error messages.

- The following events are defined on the forms:

Event	Description
COMPUTE	A normal program event that processes data the user entered on the form. For Select Form, the COMPUTE event checks whether the user entered *r* or *c* and then displays the appropriate form. For the other forms, COMPUTE calculates the area and redisplays the form.
BACK	An abnormal program event that takes the user back to a previous environment. For Select Form, the BACK event stops the program. For the other forms, BACK returns the user to Select Form.
QUIT	An abnormal program event that stops the program.

Form Definition Decks

When the designer creates the three forms (by writing a program or using Screen Design Facility), a form definition record is created with each form. For the example program, the programmer copies the following form definition decks placed by the designer on an SCU library. The library in this example is named EXAMPLE_SOURCE_LIBRARY.

The SELECT deck:

```
CHARACTER SELECT*41
CHARACTER XSELEC(41)
EQUIVALENCE (SELECT,XSELEC(1))
CHARACTER MESSAG*40
EQUIVALENCE (XSELEC(1),MESSAG)
CHARACTER OBJECT*1
EQUIVALENCE (XSELEC(41),OBJECT)
```

The RECTAN deck:

```
CHARACTER RECTAN*64
CHARACTER XRECTA(64)
EQUIVALENCE (RECTAN,XRECTA(1))
INTEGER SIDE (2)
EQUIVALENCE (XRECTA(1),SIDE(1))
INTEGER AREA
EQUIVALENCE (XRECTA(17),AREA)
CHARACTER MESSAG*40
EQUIVALENCE (XRECTA(25),MESSAG)
```

The CIRCLE deck:

```
CHARACTER CIRCLE*56
CHARACTER XCIRCL(56)
EQUIVALENCE (CIRCLE,XCIRCL(1))
REAL AREA
EQUIVALENCE (XCIRCL(1),AREA)
REAL RADIUS
EQUIVALENCE (XCIRCL(9),RADIUS)
CHARACTER MESSAG*40
EQUIVALENCE (XCIRCL(17),MESSAG)
```

Example FORTRAN Program

This FORTRAN program calls the forms and executes the events described in the previous sections. The program is in the SCU deck named FORTRAN_COMPUTE_OBJECT_AREA. To run the example program, see the Screen Formatting examples in the Examples online manual.

```
      PROGRAM COMPUT (OUTPUT, TAPE2=OUTPUT)

* Copy definitions for Screen Formatting subroutines.

*COPY FDP$FORTRAN_ALIASES

* Copy variables for select form.

*COPY select

      INTEGER IFORM, ISFORM, ICFORM, IRFORM, ISTAT,IVSTAT
      INTEGER ISX,ISY,IFX,IFY,IET,IOCCUR,ICP,IOT,IOX,IOY
      CHARACTER*31 FNAME, ENAME, ONAME, VNAME, DNAME
      CHARACTER*1 NORMAL, LAST

* Open all forms used by the program
* and assign form identifiers.

      FNAME='SELECT'
      CALL FDOPEN (FNAME, ISFORM, ISTAT)
      CALL CHECKS ('Open failed on form select', ISTAT)

      FNAME='CIRCLE'
      CALL FDOPEN (FNAME, ICFORM, ISTAT)
      CALL CHECKS ('Open failed on form circle', ISTAT)

      FNAME='RECTAN'
      CALL FDOPEN (FNAME, IRFORM, ISTAT)
      CALL CHECKS ('Open failed on form rectangle', ISTAT)

* Add select form to list scheduled for display.

      CALL FDADD (ISFORM, ISTAT)
      CALL CHECKS ('Add failed on form select', ISTAT)
```

```
* Update screen and accept user terminal entry
* for object; display all added forms.

   20 CALL FDREAD (ISTAT)
      CALL CHECKS ('Read failed on form select', ISTAT)

* Get screen event(s) that determine next actions.

      CALL FDGETE (ENAME,NORMAL,ISX,ISY,IFORM,IFX,IFY,IET,
     - ONAME,IOCCUR,ICP,IOT,IOX,IOY,LAST,ISTAT)
      CALL CHECKS ('Get event failed on form select', ISTAT)

      IF (ENAME .NE. 'COMPUTE') THEN

* Stop program on QUIT or BACK event.

        GO TO 30
      END IF

* Transfer object variable from form to program.

      VNAME = 'OBJECT'
      CALL FDGETS (ISFORM, VNAME, 1, OBJECT, IVSTAT, ISTAT)
      CALL CHECKS
     -  ('Get string variable failed on form select', ISTAT)

* If terminal user entered invalid data, display
* error message and ask for another entry.

      IF(IVSTAT .NE. 0) THEN
        CALL DISMES ('Type r or c.', ISFORM)
        GO TO 20
      END IF

      IF (OBJECT .EQ. 'R') THEN

* Remove select form and compute area of rectangle.


        CALL FDDEL (ISFORM, ISTAT)
        CALL CHECKS ('Delete failed on form select', ISTAT)
        CALL COMPR (ENAME, IRFORM)
        GO TO 25
      END IF
```

```
      IF (OBJECT .EQ. 'C') THEN

* Remove select form and compute area of circle.

        CALL FDDEL (ISFORM, ISTAT)
        CALL CHECKS ('Delete failed on form select', ISTAT)
        CALL COMPC (ENAME, ICFORM)
        GO TO 25
      END IF

* If terminal user entered invalid value for object,
* display error message and ask for another entry.


      CALL DISMES ('Type r or c.', ISFORM)
      GO TO 20

* Process event from rectangle form or circle form.

   25 IF(ENAME .EQ. 'QUIT') THEN
        GO TO 30
      END IF

* A BACK event occurred on rectangle form or circle form;
* display select form in original state.


      CALL FDRESF (ISFORM, ISTAT)
      CALL CHECKS ('Reset failed on form select', ISTAT)

      CALL FDADD (ISFORM, ISTAT)
      CALL CHECKS ('Add failed on form select', ISTAT)
      GO TO 20
```

```
* Close all forms.

   30 CALL FDCLOS (ISFORM, ISTAT)
      CALL CHECKS ('Close failed on form select', ISTAT)

      CALL FDCLOS (ICFORM, ISTAT)
      CALL CHECKS ('Close failed on form circle', ISTAT)

      CALL FDCLOS (IRFORM, ISTAT)
      CALL CHECKS ('Close failed on form rectangle', ISTAT)

      STOP
      END

      SUBROUTINE CHECKS (MESSAG, ISTAT)

* Check Screen Formatting subroutine call status.

      INTEGER ISTAT
      CHARACTER*(*) MESSAG
    5 FORMAT (1X, A, ', status = ',I4)

      IF(ISTAT .NE. 0) THEN
        WRITE (2,5) MESSAG, ISTAT
        STOP
      END IF

      RETURN
      END

      SUBROUTINE DISMES (MESSAG, IFORM)

* Display message for variable status errors.

      INTEGER IFORM, IVSTAT, ISTAT
      CHARACTER*31 VNAME
      CHARACTER*(*) MESSAG
```

```
*COPY FDP$FORTRAN_ALIASES

      VNAME='MESSAG'
      CALL FDREPS (IFORM, VNAME, 1, MESSAG, IVSTAT, ISTAT)
      CALL CHECKS ('Replace string failed on message', ISTAT)
      RETURN
      END

      SUBROUTINE COMPC (ENAME, ICFORM)

* Subroutine to  compute area for circle.

*COPY FDP$FORTRAN_ALIASES

* Copy variables for circle form.

*COPY circle

      INTEGER IFORM, ISTAT,IVSTAT, ICFORM
      INTEGER ISX,ISY,IFX,IFY,IET,IOCCUR,ICP,IOT,IOX,IOY
      CHARACTER*31 ENAME, ONAME, VNAME
      CHARACTER*1 NORMAL, LAST

* Display circle form in original state.

      CALL FDRESF (ICFORM, ISTAT)
      CALL CHECKS ('Reset failed on form circle', ISTAT)

      CALL FDADD (ICFORM, ISTAT)
      CALL CHECKS ('Add failed on form circle', ISTAT)

* Update screen and get radius from terminal user entry.

    5 CALL FDREAD (ISTAT)
      CALL CHECKS ('Read failed on form circle ', ISTAT)

      CALL FDGETE (ENAME,NORMAL,ISX,ISY,IFORM,IFX,IFY,IET,
     - ONAME,IOCCUR,ICP,IOT,IOX,IOY,LAST,ISTAT)
      CALL CHECKS ('Get event failed on form circle', ISTAT)

      IF (ENAME .NE. 'COMPUTE') THEN
        CALL FDDEL (ICFORM, ISTAT)
        CALL CHECKS ('Delete failed on form circle', ISTAT)
        RETURN
      END IF
```

```
* Transfer terminal user entry for radius to program.

      VNAME = 'RADIUS'
      CALL FDGETR (ICFORM, VNAME, 1, RADIUS, IVSTAT, ISTAT)
      CALL CHECKS
     -('Get real variable failed on form circle', ISTAT)
      IF(IVSTAT .NE. 0) THEN
        CALL DISMES ('Type valid value for radius.', ICFORM)
        GO TO 5
      END IF

* Compute area of circle and display it.

      AREA=3.15*(RADIUS**2)

      VNAME = 'AREA'
      CALL FDREPR (ICFORM, VNAME, 1, AREA, IVSTAT, ISTAT)
      CALL CHECKS
     -('Replace real variable failed on form circle', ISTAT)
      IF(IVSTAT .NE. 0) THEN

* The area value could not be displayed using
* output format defined for form.
* Revise the form or the program to accommodate
* size of number.

        CALL DISMES ('Type valid value for radius.', ICFORM)
        GO TO 5
      END IF

* Blank error message in case previously displayed.

      CALL DISMES (' ', ICFORM)

* Process next user entry.

      GO TO 5
      END

      SUBROUTINE COMPR (ENAME, IRFORM)

* Subroutine to compute area of rectangle.

*COPY FDP$FORTRAN_ALIASES
```

```
* Copy variables for rectangle form.

*COPY rectan

      INTEGER IFORM, ISTAT,IVSTAT,IRFORM
      INTEGER ISX,ISY,IFX,IFY,IET,IOCCUR,ICP,IOT,IOX,IOY
       CHARACTER*31 ENAME, ONAME, VNAME, DNAME
      CHARACTER*1 NORMAL, LAST

       DNAME='ERROR'

* Display rectangle form in original state.

      CALL FDRESF (IRFORM, ISTAT)
      CALL CHECKS ('Reset failed on form rectangle', ISTAT)

      CALL FDADD (IRFORM, ISTAT)
      CALL CHECKS ('Add failed on form rectangle', ISTAT)

* Update screen and get terminal user entry
* for rectangle height and width.

    5 CALL FDREAD (ISTAT)
      CALL CHECKS ('Read failed on form rectangle', ISTAT)

      CALL FDGETE (ENAME,NORMAL,ISX,ISY,IFORM,IFX,IFY,IET,
     - ONAME,IOCCUR,ICP,IOT,IOX,IOY,LAST,ISTAT)
      CALL CHECKS ('Get event failed on form rectangle', ISTAT)

* If abnormal event (BACK or QUIT) occurs, return to caller.

      IF (ENAME .NE. 'COMPUTE') THEN
        CALL FDDEL (IRFORM, ISTAT)
        CALL CHECKS ('Delete failed on form rectangle', ISTAT)
        RETURN
      END IF
```

```
* Remove any previous error indications.

      VNAME = 'SIDE'
      CALL FDRESO (IRFORM, VNAME, 1, ISTAT)
      CALL CHECKS
     -('Reset object failed on form rectangle', ISTAT)
      CALL FDRESO (IRFORM, VNAME, 2, ISTAT)
      CALL CHECKS
     -('Reset object failed on form rectangle', ISTAT)
      CALL DISMES (' ', IRFORM)

* Transfer height value from form to program.

      VNAME = 'SIDE'
      CALL FDGETI (IRFORM, VNAME, 1,SIDE (1), IVSTAT, ISTAT)
      CALL CHECKS
     -('Get integer variable failed on form rectangle', ISTAT)

* If data invalid, move cursor to height value
* and display error message.

      IF(IVSTAT .NE. 0) THEN
        CALL FDSETC (IRFORM, VNAME, 1, 1, ISTAT)
        CALL CHECKS
     -('Set cursor failed on form rectangle', ISTAT)
        CALL FDSETO (IRFORM, VNAME, 1, DNAME, ISTAT)
        CALL CHECKS
     -('Set object failed on form rectangle', ISTAT)
        CALL DISMES ('Type valid value for height.', IRFORM)
        GO TO 5
      END IF

* Transfer width value from form to program.

       CALL FDGETI (IRFORM, VNAME, 2, SIDE(2), IVSTAT, ISTAT)
      CALL CHECKS
     -('Get integer variable failed on form rectangle', ISTAT)
```

```
* If data invalid, move cursor to width value and display
* error message.

      IF(IVSTAT .NE. 0) THEN
        CALL FDSETC (IRFORM, VNAME, 2, 1, ISTAT)
        CALL CHECKS
     -('Set cursor failed on form rectangle', ISTAT)
         CALL FDSETO (IRFORM, VNAME, 2, DNAME, ISTAT)
        CALL CHECKS
     -('Set object failed on form rectangle', ISTAT)
        CALL DISMES ('Type valid value for width.', IRFORM)
        GO TO 5
      END IF

* Compute area of rectangle and display it.

      AREA=SIDE(1)*SIDE(2)

      VNAME = 'AREA'
      CALL FDREPI (IRFORM, VNAME, 1, AREA, IVSTAT, ISTAT)
      CALL CHECKS
     -('Replace integer variable failed on form rectangle',
     -ISTAT)
      IF(IVSTAT .NE. 0) THEN

* Area value could not be displayed using
* output format defined for form.
* Revise the form or the program to accommodate
* size of number.

        CALL DISMES ('Format cannot display area.', IRFORM)
        GO TO 5
      END IF

* Process next user entry.

      GO TO 5
      END
```

Expanding and Compiling a Program

Programs using Screen Formatting use common decks and form definition records that reside outside the main program. To manage the source text for this type of program, put the program in one or more SCU decks. This allows you to update individual parts of a program and to use forms in more than one program without duplicating code.[3]

To expand and compile a program maintained in SCU decks:

1. Expand the deck containing the main program.

2. Compile the expanded program.

3. Put the compiled program on an object library.

A procedure for compiling and expanding a program is shown in the following example. (The example is based on the example program and form definition records described earlier. The example shows how to place decks on library EXAMPLE_SOURCE_LIBRARY.)

The procedure calls SCU to expand the SCU directives contained in the program. For this expansion, you must include the $SYSTEM.CYBIL.OSF$PROGRAM_INTERFACE library as an alternate base. The program is then compiled and put on an object library.

```
PROCEDURE fortran_compile_deck, forcd (
  deck,  d: name = $required
  status)

  source_code_utility
    use_library base=example_source_library result=$null
    expand_deck deck=deck ..
    compile=$local.compile ..
    alternate_base=$system.cybil.osf$program_interface
  quit

  fortran input=$local.compile ..
    list=$local.listing runtime_checks=all  ..
     binary_object=$local.lgo  ..
    debug_aids=dt
```

3. For information on SCU, see the NOS/VE Source Code Management manual.

```
    create_object_library
      add_module library=example_object_library
      combine_module library=$local.lgo
      generate_library library=example_object_library.$next
    quit

  PROCEND fortran_compile_deck
```

To use the procedure, put it on library EXAMPLE_OBJECT_LIBRARY and then add the library to your command list (using the CREATE_COMMAND_LIST_ENTRY command). You can execute the procedure by entering:

```
/fortran_compile_deck deck=fortran_compute_object_area
```

The compiled program is now also on library EXAMPLE_OBJECT_LIBRARY.

For more information on writing and using procedures, see the NOS/VE System Usage manual.

Helping the User Start the Application

The complete application consists of your program and the forms created by the designer. To integrate the forms with your program, you must:

- Create a procedure that gives users access to the object library containing the forms and program.
- Ensure that the user's terminal environment is set up properly to use the forms (in most instances, by creating a user prolog).
- Ensure that users select the correct natural language.
- Ensure that users know how to start the application.

Creating a User Procedure

To give the user access to the object library containing the forms:

1. Write a NOS/VE procedure from which the user starts the application.
2. Place the procedure on the library that contains the compiled program.

For example, the following procedure executes the application that uses the starting procedure COMPUT on library EXAMPLE_OBJECT_LIBRARY. The other libraries accessed by the program are $SYSTEM.FDF$LIBRARY and $SYSTEM.TDU.TERMINAL_DEFINITIONS. Users must have these libraries available in order for the program to call the Screen Formatting subroutines.

```
PROCEDURE fortran_compute_area, forca (
  status)

  execute_task ..
    library=(example_object_library,$system.fdf$library,..
    $system.tdu.terminal_definitions) ..
    starting_procedure=comput

PROCEND fortran_compute_area
```

Creating a User Prolog

To ensure that the users' terminal environment is set up properly to use the forms, make sure they set the following terminal characteristics before they execute the procedure:

Characteristic	Description
Terminal model	Identifies the terminal to NOS/VE.
Attention character	Provides a character users can enter to interrupt the application.
Hold messages	Tells the network to hold all network messages until the user stops the application. Otherwise, a computer operator message may overwrite a form while a user is entering data, confusing the user.

In most instances, users should set up their terminal for the entire terminal session in their user prologs. The example below does the following:

- Identifies a Digital Equipment Corporation VT220 terminal to the system.
- Chooses the exclamation point as a way to interrupt the program.
- Holds all messages from a NAMVE/CDCNET network.
- Sets up the way the terminal uses the exclamation point to interrupt the program.

The users add the following commands to their user prologs:

```
change_terminal_attributes terminal_model=dec_vt220 ..
  attention_character='!' ..
  status_action=hold
change_term_conn_defaults attention_character_action=1
change_connection_attributes terminal_file_name=input aca=1
change_connection_attributes terminal_file_name=output aca=1
change_connection_attributes terminal_file_name=command aca=1
```

For a further explanation of how to interrupt a screen application during an interactive session, and what commands to use for networks other than NAMVE/CDCNET, see the NOS/VE System Usage manual.

Selecting a Natural Language

To ensure that users receive messages in the correct natural language, have them add the CHANGE_NATURAL_LANGUAGE command to their prologs. Because the default language is US_ENGLISH and all messages returned by Screen Formatting are in this language, have users include this command only when you have changed messages to another language.

Changing messages to other languages is described in the NOS/VE Object Code Management manual. The CHANGE_NATURAL_LANGUAGE command is described in the NOS/VE System Usage manual.

Starting the Application

To start the application, the users enter:

```
/create_command_list_entry e=example_object_library
/fortran_compute_area
```

When finished with the application, the users remove the object library from their command lists:

```
/delete_command_list_entry e=example_object_library
```

FORTRAN Subroutine Calls for Interacting with Forms

The following sections describe the FORTRAN subroutine calls to Screen Formatting modules. For each subroutine, there is a purpose description, input format, list of parameters and their types, and pertinent remarks.

The FORTRAN program calls Screen Formatting subroutines that allow a user to interact with forms. These subroutines are external routines that reside on the library called $SYSTEM.FDF$LIBRARY. This library must be in the user's program library list in order to execute the program.

A subroutine name is an alias that is defined by the deck FDP$FORTRAN_ALIASES. The SCU directive *COPY FDP$FORTRAN_ALIASES must be included for each application subroutine that calls a Screen Formatting subroutine. See appendix F for a list of aliases.

When checking the status of subroutines, you must check the ISTAT parameter and also, when present, check the IVSTAT parameter. If the value of ISTAT is zero, you can process output from the subroutine. However, if there is also a IVSTAT parameter, check its value before using variable output from the subroutine. The variable status is independent of the status for the subroutine.

Adding a Form

Purpose FDADD schedules a form for display on the application user's screen.

Format **CALL FDADD (iform, istat)**

Parameters iform {input}

The identifier established when the form was opened. Include the following type statement:

```
INTEGER iform
```

istat {output}

The variable that indicates the results of the subroutine. The following values can be returned:

Value	Meaning
0	Routine completed successfully.
7	No space is available.
9	Form identifier is invalid.
36	System error occurred.
39	Form is pushed.
70	Form is already added.
131	Form is too large for screen.
145	Data value is bad.

Include the following type statement:

```
INTEGER istat
```

Remarks

- When you call either the FDREAD or FDSHOW subroutine, Screen Formatting displays the added form on the terminal screen. The added form is placed on top of other forms occupying the same area on the screen.
- When displayed, each form that is added operates independently from other forms that have been added. When a user executes a normal event, Screen Formatting validates and updates only those variables on the form associated with the event. To have forms share events, see Combining Forms later in this section.
- Before you add a form, you must open it.
- You cannot add a pushed form.

Changing Table Size

Purpose FDCHAT changes the size of the table during program execution.

Format **CALL FDCHAT (iform, tname, isize, istat)**

Parameters iform {input}

The identifier established when the form was opened. Include the following type statement:

INTEGER iform

tname {input}

The name of the table to change in size. Include the following type statement:

CHARACTER*31 tname

isize {input}

The size of the table. While this subroutine is in effect, Screen Formatting limits the number of stored occurrences allowed for a table to the value you specify on this parameter. How many occurrences are displayed at one time depends on the number of visible occurrences defined in the form.

If you specify zero for the table size, no occurrences appear on the form.

Include the following type statement:

INTEGER isize

istat {output}

The variable that indicates the results of the subroutine. The following values can be returned:

Value	Meaning
0	Routine completed successfully.
7	No space is available.
9	Form identifier is invalid.
37	Table name is invalid.
39	Form is pushed.
40	Table name is unknown.
145	Data value is bad.
151	Table size is invalid.

Include the following type statement:

```
INTEGER istat
```

Remarks

- The table must be present in an open form.
- The size limitation remains in effect until the next time you call the FDCHAT subroutine.
- The maximum size for a table is identified by the form as the maximum number of stored occurrences. If you specify a table size larger than the maximum, you receive an error message (table size is invalid).

Examples

The following examples describe how changing the size of a table affects the application user. On the form, the table's specifications are a maximum of 20 stored occurrences, of which 6 occurrences can be visible at one time.

- If you specify a table size of 10, Screen Formatting displays 6 occurrences and allows the application user to page to the 10th occurrence.
- If you specify a table size of 4, Screen Formatting displays 4 occurrences and does not allow the application user to page.

Closing a Form

Purpose

FDCLOS releases resources used to process a form and deletes the form from the list scheduled for display.

Format

CALL FDCLOS (iform, istat)

Parameters

iform {input}

The identifier established when the form was opened. Include the following type statement:

INTEGER iform

istat {output}

The variable that indicates the results of the subroutine. The following values can be returned:

Value	Meaning
0	Routine completed successfully.
7	No space is available.
9	Form identifer is invalid.
39	Form is pushed.
145	Data value is bad.

Include the following type statement:

INTEGER istat

Remarks

- When the program calls either the FDREAD or FDSHOW subroutine, Screen Formatting removes the closed form from the terminal screen as a result of calling this procedure.
- Before you can close a form, you must open it.
- You cannot close a pushed form.

Combining Forms

Purpose

FDCOM combines a form with a previously added form and schedules the combined form for display on the terminal screen.

Format

CALL FDCOM (iaform, icform, istat)

Parameters

iaform {input}

The identifier for this instance of the previously added form. Include the following type statement:

```
INTEGER iaform
```

icform {input}

The identifier for the form you are combining with the previously added form. Include the following type statement:

```
INTEGER icform
```

istat {output}

The variable that indicates the results of the subroutine. The following values can be returned:

Value	Meaning
0	Routine completed successfully.
7	No space is available.
9	Form identifer is invalid.
39	Form is pushed.
70	Form is already added.
131	Form is too large for screen.
145	Data value is bad.
150	Form is already combined.
152	Form is not added.

Include the following type statement:

```
INTEGER istat
```

Remarks

- You cannot combine a pushed form.
- The combined form inherits the event definitions of the previously added form.
- Before you combine a form with a previously added form, you must open both forms.
- When the programs calls either the FDREAD or FDSHOW subroutine, Screen Formatting displays the combined form. The combined form is placed on top of other forms occupying the same area on the screen.
- When the application user executes an event to return normally to the program, Screen Formatting updates all program variables associated with both the added and combined forms.
- To combine several forms with a previously added form, call this subroutine more than once.

Deleting a Form

Purpose FDDEL deletes the form from the list of forms scheduled for display.

Format **CALL FDDEL (iform, istat)**

Parameters iform {input}

The identifier established when the form was opened. Include the following type statement:

INTEGER iform

istat {output}

The variable that indicates the results of the subroutine. The following values can be returned:

Value	Meaning
0	Routine completed successfully.
7	No space is available.
9	Form identifer is invalid.
39	Form is pushed.
54	Form is not scheduled for display.
145	Data value is bad.

Include the following type statement:

INTEGER istat

Remarks

- When the program calls either the FDREAD or FDSHOW subroutine, Screen Formatting removes the deleted form from the terminal screen and replots any forms uncovered by the deleted form.
- When you add a form (FDADD) again that you previously deleted, the data in the form is retained.
- Before you delete a form, you must open it.
- You cannot delete a pushed form.
- If the form was added and has any combined forms associated with it, the combined forms are also deleted.
- When you delete a combined form, only that form is deleted. Areas covered by the combined form are replotted after the combined form is deleted.

Getting an Integer Variable

Purpose FDGETI gets the value the user entered on a form for an integer variable and transfers it to the program.

Format **CALL FDGETI (iform, vname, ioccur, ivar, ivstat, istat)**

Parameters

iform {input}

The identifier established when the form was opened. Include the following type statement:

INTEGER iform

vname {input}

The name of the variable to get and transfer to the program. The name was defined when the form was created.

ioccur {input}

The occurrence of the variable name. Include the following type statement:

INTEGER ioccur

ivar {output}

The integer variable that Screen Formatting generates automatically in the form definition record. If you do not want to use the automatically generated variable, include the following type statement:

INTEGER ivar

ivstat {output}

The condition that gives you the status of the variable. The following values can be returned:

Value	Meaning
0	No error occurred on the variable.
3	The user entered data that is not a valid integer.
5	The user entered data that does not match the defined program data type.
7	User entered an integer that is too large.

Include the following type statement:

INTEGER ivstat

istat {output}

The variable that indicates the subroutine results. The following values can be returned:

Value	Meaning
0	Routine completed successfully.
7	No space is available.
9	Form identifer is invalid.
11	Variable name is unknown.
36	System error exists.
38	Variable name is invalid.
91	Occurrence is unknown.
145	Data value is bad.
147	Variable type is wrong.

Include the following type statement:

INTEGER istat

Remarks

- Before you get an integer variable, you must open its form. If you get the variable after opening the form and before reading or replacing the variable on the form, the program returns the initial value specified by the form designer.

- If the form designer specifies data validation rules and error processing to display an error message or form, the program does not need to look at the variable status parameter.

  If the form designer specifies data validation rules and no error processing, the program must look at the variable status parameter.

  If the form designer specifies no data validation rules, the program must look at the variable status parameter.

Getting the Next Event

Purpose FDGETE gets the event resulting from the most recent FDREAD subroutine.

Format **CALL FDGETE (ename, normal, isx, isy, iform, ifx, ify, iet, oname, ioccur, icp, iot, iox, ioy, last, istat)**

Parameters ename {output}

A data name to receive the application user's event. Include the following type statement:

CHARACTER*31 ename

normal {output}

A data name to receive the event normal indication. If the event is normal, T is returned. If the event is not normal, F is returned. Include the following type statement:

CHARACTER*1 normal

isx {output}

A data name to receive the x position of the event on the screen. The character position in the upper left corner of the screen is 1; the x position increases by 1 for each character you count from left to right. Include the following type statement:

INTEGER isx

isy {output}

A data name to receive the y position of the event on the screen. The character position in the upper left corner of the screen is 1; the y position increases by 1 for each character you count from top to bottom. Include the following type statement:

INTEGER isy

iform {output}

The variable that returns the instance of the form for the event. Include the following type statement:

INTEGER iform

ifx {output}

A data name to receive the x position of the event on the form. The character in the upper left corner of the form is 1; the x position increases by 1 for each character you count from left to right. Include the following type statement:

```
INTEGER ifx
```

ify {output}

A data name to receive the y position of the event on the form. The character in the upper left corner of the form is 1; the y position increases by 1 for each character you count from top to bottom. Include the following type statement:

```
INTEGER ify
```

iet {output}

The event type. The following values are possible:

Value	Meaning
0	The event occurred on an area of a form containing no object.
1	The event occurred on a form object.

Include the following type statement:

```
INTEGER iet
```

oname {output}

When event type is 1, the variable returns a value giving the name of the object where the event occurred. Include the following type statement:

```
CHARACTER*31 oname
```

ioccur {output}

When event type is 1, the variable returns a value giving the occurrence of the object name. Include the following type statement:

```
INTEGER ioccur
```

icp {output}

When event type is 1, the variable returns a value giving the character position within the object where the event occurred. The first character position is 1. Include the following type statement:

INTEGER icp

iot {output}

When event type is 1, the variable indicates the type of object on which the event occurred. The following values are possible:

Value	Object Type
0	Box
1	Constant text
2	Constant box
3	Line
5	Variable text
6	Variable box

Include the following type statement:

INTEGER iot

iox {output}

When event type is 1, the value returned is the x origin position of the object. The character in the upper left corner of the form is 1; the x position increases by 1 for each character you count from left to right. Include the following type statement:

INTEGER iox

ioy {output}

When event type is 1, the value returned is the y origin position of the object. The character in the upper left corner of the form is 1; the y position increases by 1 for each character you count from top to bottom. Include the following type statement:

INTEGER ioy

last {output}

Indicates whether this is the last event The following values are possible:

Value	Meaning
T	This is the last event.
F	This is not the last event.

Include the following type statement:

CHARACTER*1 last

istat {output}

The variable that indicates the results of the subroutine. The following values can be returned:

Value	Meaning
0	Routine completed successfully.
145	Data value is bad.

Include the following type statement:

INTEGER istat

Remarks

The FDREAD subroutine deletes existing events. If the event is normal, Screen Formatting updates the variables in the added and combined forms containing the event. Later, you can request the transfer of these variables to program storage. If the event is abnormal, Screen Formatting does not update or validate variables.

Getting a Real Variable

Purpose FDGETR gets a value the user entered on a form for a real variable and transfers it to the program.

Format **CALL FDGETR (iform, vname, ioccur, var, ivstat, istat)**

Parameters iform {input}

The identifier established when the form was opened. Include the following type statement:

```
INTEGER iform
```

vname {input}

The name of the variable to get. The name was defined when the form was created.

ioccur {input}

The occurrence of the variable name. Include the following type statement:

```
INTEGER ioccur
```

var {output}

The value of the real variable that Screen Formatting generates automatically in the form definition record. If you do not want to use the automatically generated variable, include the following type statement:

```
REAL var
```

ivstat {output}

The condition that gives you the status of the variable. The following values are possible:

Value	Meaning
0	No error occurred on the variable.
2	The user entered data that is within the range of real numbers defined for the variable.
5	The user entered data that does not correspond to the defined data type.
7	The user entered a number too large to be converted to the defined real program type.
9	The user entered an exponent that is too large.
10	User entered an exponent that is too small.
11	User entered an indefinite number.

Include the following type statement:

```
INTEGER ivstat
```

istat {output}

The variable that indicates the results of the subroutine. The following values are possible:

Value	Meaning
0	Routine completed successfully.
7	No space is available.
9	Form identifier is invalid.
11	Variable name is unknown.
36	System error exists.
38	Variable name is invalid.
91	Occurrence is unknown.
145	Data value is bad.
147	Variable type is wrong.

Include the following type statement:

```
INTEGER istat
```

Remarks

- Before you get a real variable, you must open the form on which the user enters the value. If you get the variable after opening the form and before reading or replacing the variable on the form, the program returns the initial value specified by the form designer.

- If the form designer specifies data validation rules and error processing to display an error message or form, your program does not need to look at the variable status parameter.

  If the form designer specifies data validation rules and no error processing, the program must look at the variable status parameter.

  If the form designer specifies no data validation rules, the program must look at the variable status parameter.

Getting a Record

Purpose FDGET transfers the values of the form record to the program record.

Format **CALL FDGET (iform, record, ivstat, istat)**

Parameters iform {input}

The identifier established when the form was opened. Include the following type statement:

```
INTEGER iform
```

record {output}

The name of the record that contains working storage information for the form. When the form is created, Screen Formatting generates the type statements in this record. It is the program work area for the variables used on the form.

ivstat {output}

the condition that gives you the status of the variable. The following values are possible:

Value	Meaning
0	No error occurred on the variable.
1	The user entered data that does not match the strings defined for the variable.
2	The user entered data that is not within the range of real numbers defined for the variable.
3	The user entered data that is not within the range of integer numbers defined for the variable.
5	The user entered data that does not correspond to the defined data type.
7	User entered a number that is too large to be converted to the defined real or integer data type.
9	The user entered an exponent that is too large.

Value	Meaning
10	The user entered an exponent that is too small.
11	The user entered an indefinite number.
12	The user entered an infinite number.

Include the following type statement:

INTEGER ivstat

istat {output}

The variable that indicates the results of the subroutine. The following values are possible:

Value	Meaning
0	Routine completed successfully.
9	Form identifer is invalid.
14	Work area is invalid.
36	System error exists.
52	Form has no variable.
145	Data value is bad.

Include the following type statement:

INTEGER istat

Remarks

- Before you get a record for a form, you must open the form. If you get the record after opening the form and before reading or replacing the record, the program returns the initial value specified by the form designer.

- If the form designer specifies data validation rules and error processing to display an error message or form, your program does not need to look at the variable status parameter.

  If the form designer specifies data validation rules and no error processing, the program must look at the variable status parameter.

  If the form designer specifies no data validation rules, the program must look at the variable status parameter.

Getting a String Variable

Purpose FDGETS gets a value the user entered on a form for a string variable and transfers it to the program.

Format **CALL FDGETS (iform, vname, ioccur, cvar, ivstat, istat)**

Parameters iform {input}

The identifier established when the form was opened. Include the following type statement:

```
INTEGER iform
```

vname {input}

The name of the variable to get. The name was defined when the form was created.

ioccur {input}

The occurrence of the variable name. Include the following type statement:

```
INTEGER ioccur
```

cvar {output}

The variable that Screen Formatting generates automatically in the form definition record. The form definition record defines the variable. If you do not want to use the automatically generated variable, include the following type statement (n is the number of characters in the variable):

```
CHARACTER*n
```

ivstat {output}

The condition that gives you the status of the variable. The following values are possible:

Value	Meaning
0	No error occurred on the variable.
1	The user entered data that does not match the strings defined for variable.
15	The storage length of the parameter variable is not long enough.

Include the following type statement:

```
INTEGER ivstat
```

istat {output}

The variable that indicates the results of the subroutine. The following values are possible:

Value	Meaning
0	Routine completed successfully.
7	No space is available.
9	Form identifer is invalid.
11	Variable name is unknown.
36	System error exists.
38	Variable name is invalid.
91	Occurrence is unknown.
145	Data value is bad.
147	Variable type is wrong.

Include the following type statement:

```
INTEGER istat
```

Remarks

- Before you get a string variable, you must open the form on which the user enters the value. If you get the variable after opening the form and before reading or replacing the variable on the form, the program returns the initial value specified by the form designer.

- If the form designer specifies data validation rules and error processing to display an error message or form, your program does not need to look at the variable status parameter.

  If the form designer specifies data validation rules and no error processing, the program must look at the variable status parameter.

  If the form designer specifies no data validation rules, the program must look at the variable status parameter.

Opening a Form

Purpose FDOPEN locates a form and prepares it for use by the program.

Format **CALL FDOPEN (fname, iform, istat)**

Parameters fname {input}

The name of the form you want to open. Include the following type statement:

```
CHARACTER*31 fname
```

iform {input-output}

The form identifier established for the form. Other Screen Formatting subroutines use this identifier when referencing the form. Include the following type statement:

```
INTEGER iform
```

istat {output}

The variable that indicates the results of the subroutine. The following values are possible:

Value	Meaning
0	Routine completed successfully.
5	Form name is unknown.
7	No space is available.
9	Form indentifier is invalid.
26	Form name is invalid.
36	System error exists.
100	Terminal is not defined.
136	Form is not ended.
139	Form is already open.
141	Form requires conversion.
145	Data value is bad.

Include the following type statement:

```
INTEGER istat
```

Remarks

- Screen Formatting locates a form as follows:
  - If the form name is blank, Screen Formatting assumes that the form identifier specifies the required dynamically created form.
  - If the form name is not blank, Screen Formatting searches the list of ended dynamically created forms.
  - If the form name is not blank and is not in the list of ended dynamically created forms, Screen Formatting searches the command library list to find the form name on the object code libraries. (You specify the order in which Screen Formatting searches the list using the NOS/VE command CREATE_COMMAND_LIST_ENTRY).
- Executing FDP$XOPEN_FORM does not display the form on the screen.
- The form identifier that FDOPEN returns identifies the instance of open for a form. Forms dynamically created have only one instance of open. Forms stored on object code libraries can have more than one instance of open. For each instance of open, Screen Formatting maintains the working environment (current value of variables and their display attributes) of the form.

Popping a Form

Purpose FDPOP deletes forms scheduled (added or combined) since the last FDPUSH subroutine.

Format **CALL FDPOP (istat)**

Parameters istat {output}

The variable that indicates the results of the subroutine. The following values are possible:

Value	Meaning
0	Routine completed successfully.
42	No forms are available to pop.
145	Data value is bad.

Include the following type statement:

INTEGER istat

Remarks Events associated with the last list of pushed forms become active.

Positioning a Form

Purpose FDPOS schedules moving a form to a new location. Using this subroutine, you can define a form at one location and display it at another location, or you can move a form from where it is currently displayed to a new location.

Format **CALL FDPOS (iform, isx, isy, istat)**

Parameters iform {input}

The form identifier established when the form was opened. Include the following type statement:

INTEGER iform

isx {input}

The x position on the screen. The character position in the upper left corner of the screen is 1, and the x position increases by 1 for each character you count from left to right. Include the following type statement:

INTEGER isx

isy {input}

The y position on the screen. The character position in the upper left corner of the screen is 1, and the y position increases by 1 for each character you count from top to bottom. Include the following type statement:

INTEGER isy

istat {output}

The variable that indicates the results of the subroutine. The following values are possible:

Value	Meaning
0	Routine completed successfully.
7	No space is available.
9	Form identifer is invalid.
36	System error exists.
39	Form is pushed.
54	Form is not scheduled
131	Form is too large for screen.
145	Data value is bad.

Include the following type statement:

INTEGER istat

Remarks

- When the program calls either the FDREAD or FDSHOW subroutine, Screen Formatting displays the form on the screen at the position specified in the call to FDPOS.
- If you call this subroutine while the form is displayed, the form is deleted from its current location and added at the new location. The added form lays on top of any other form occupying the same area on the screen.
- If you call this procedure before the form is displayed, the form is displayed at the specified location.
- Before you position a form, you must open it.
- You cannot position a pushed form.

Pushing a Form

Purpose FDPUSH causes Screen Formatting to record added and combined forms so you can return to them later.

Format **CALL FDPUSH (istat)**

Parameters istat {output}

The variable that indicates the results of the subroutine. The following values are possible:

Value	Meaning
0	Routine completed successfully.
46	No forms are available to push.
145	Data value is bad.

Include the following type statement:

```
INTEGER istat
```

Remarks

- Events associated with these forms are not passed to the program.
- A program cannot change or close a pushed form.
- Pushed forms are displayed on the screen. If you want newly added forms to appear on a blank screen, first add a blank form that covers the screen.

  Updates to the screen continue to show the pushed forms.
- This subroutine deactivates the events associated with forms scheduled for display (added or combined) since the last push call.

Reading Forms

Purpose FDREAD updates the terminal screen and accepts input from the application user.

Format **CALL FDREAD (istat)**

Parameters istat {output}

The variable that indicates the results of the subroutine. The following values are possible:

Value	Meaning
0	Routine completed successfully.
1	Terminal is disconnected.
36	System error exists.
104	No forms to read.
142	No events are active.
145	Data value is bad.

Include the following type statement:

```
INTEGER istat
```

Remarks

- A call to FDREAD:
  - Displays all the forms you scheduled for display and have not deleted. If you added or combined forms since the last FDREAD or FDSHOW call, it displays them for the first time.
  - Removes from the screen the forms you deleted since the last FDREAD or FDXSHOW call.
  - Updates on the screen the variables replaced since the last FDREAD or FDSHOW call.
  - Updates on the screen the objects for which display attributes were set or reset since the last FDREAD or FDSHOW call.
- Events not retrieved with the FDGETE subroutine are deleted before any input is accepted from the user.
- The FDREAD subroutine does not execute unless the forms scheduled for display contain at least one active event.

Replacing an Integer Variable

Purpose FDREPI transfers a program integer variable to Screen Formatting.

Format **CALL FDREPI (iform,vname,ioccur,ivar,ivstat,istat)**

Parameters

iform {input}

The identifier established when the form was opened. Include the following type statement:

INTEGER iform

vname {input}

The name of the integer variable to replace. The name was defined when the form was created.

ioccur {input}

The occurrence of the variable name. Include the following type statement:

INTEGER ioccur

ivar {input}

The integer variable that Screen Formatting generates automatically in the form definition record. If you do not want to use the automatically generated variable, include the following type statement:

INTEGER ivar

ivstat {output}

The condition that gives you the status of the variable. The following values are possible:

Value	Meaning
0	No error occurred on the variable.
3	The program supplied a variable that is not within the range of integer numbers defined for the variable.

Value	Meaning
7	The program supplied a value that is too large for the form variable.
14	The output format defined for the variable cannot cannot output the variable.

Include the following type statement:

INTEGER ivstat

istat {output}

The variable that indicates the results of the subroutine. The following values are possible:

Value	Meaning
0	Routine completed successfully.
7	No space is available.
9	Form indentifer is invalid.
11	Variable name is unknown.
36	System error exists.
38	Variable name is invalid.
39	Form is pushed.
91	Occurrence is unknown.
145	Data value is bad.
147	Variable type is wrong.

Include the following type statement:

INTEGER istat

Remarks

- When you call either the FDREAD or FDSHOW subroutine, Screen Formatting replaces the integer variable on the terminal screen.
- Before you replace an integer variable, you must open the form on which it is replaced.
- You cannot replace an integer variable for a pushed form.
- If the integer variable is not valid, it is not replaced.

Replacing a Real Variable

Purpose FDREPR transfers a program real variable to Screen Formatting.

Format **CALL FDREPR (iform, vname, ioccur, var, ivstat, istat)**

Parameters iform {input}

The identifier established when the form was opened. Include the following type statement:

INTEGER iform

vname {input}

The name of the real variable to replace. The name was defined when the form was created.

ioccur {input}

The occurrence of the variable name. Include the following type statement:

INTEGER ioccur

var {input}

The value of the real variable that Screen Formatting generates automatically in the form definition record. If you do not want to use the automatically generated variable, include the following type statement:

REAL var

ivstat {output}

The condition that gives you the status of the variable. The following values are possible:

Value	Meaning
0	No error occurred on the variable.
2	The value the program supplied is not within the range of real numbers defined for the variable.

Value	Meaning
7	The value the program supplied is too large for the for variable.
14	The output format defined for the variable cannot output the variable.

Include the following type statement:

INTEGER ivstat

istat {output}

The variable that indicates the results of the subroutine. The following values are possible:

Value	Meaning
0	Routine completed successfully.
7	No space is available.
9	Form identifer is invalid.
11	Variable name is unknown.
36	System error exists.
38	Variable name is invalid.
39	Form is pushed.
91	Occurrence is unknown.
145	Data value is bad.
147	Variable type is wrong.

Include the following type statement:

INTEGER istat

Remarks

- When you call either the FDREAD or FDSHOW subroutine, Screen Formatting replaces the real variable on the terminal screen.
- Before you replace a real variable, you must open the form on which it is replaced.
- You cannot replace a real variable for a pushed form.
- If the real variable is not valid, it is not replaced.

Replacing a Record

Purpose

FDREP transfers values of program variables to Screen Formatting for later display on a form.

Format

CALL FDREP (iform, record, ivstat, istat)

Parameters

iform {input}

The identifier established when the form was opened. Include the following type statement:

```
INTEGER iform
```

record {input}

The name of the record that contains working storage information for the form. When the form is created, Screen Formatting generates the type statements in this record. It is the program work area for the variables used on the form.

ivstat {output}

The condition that gives you the status of the variable.

Value	Meaning
0	No error occurred on the variable.
1	The program supplied an invalid string variable.
2	The program supplied an invalid real variable.
3	The program supplied an invalid integer variable.
7	The program supplied a number too large to be converted to the form variable size.
9	The program supplied an exponent that is too large.
10	The program supplied an exponent that is too small.
11	The program supplied an indefinite number.

Value	Meaning
12	The program supplied an infinite number.
14	The output format defined for the variable cannot output the variable.

Include the following type statement:

INTEGER ivstat

istat {output}

The variable that indicates the results of the subroutine. The following values are possible:

Value	Meaning
0	Routine completed successfully.
7	No space is available.
9	Form indentifer is invalid.
14	Work area is invalid.
39	Form is pushed.
52	Form has no variable.
145	Data value is bad.

Include the following type statement:

INTEGER istat

Remarks

- When the program calls either the FDREAD or FDSHOW subroutine, Screen Formatting replaces the variables on the terminal screen with the values stored in Screen Formatting.
- Before you replace a record, you must open the form on which the variables are replaced.
- You cannot replace a record for a pushed form.

Replacing a String Variable

Purpose FDREPS transfers a program string variable to Screen Formatting.

Format **CALL FDREPS (iform, vname, ioccur, cvar, ivstat, istat)**

Parameters **iform** {input}

The identifier established when the form was opened. Include the following type statement:

```
INTEGER iform
```

vname {input}

The name of the string variable to replace. The name was defined when the form was created.

ioccur {input}

The occurrence of the variable name. Include the following type statement:

```
INTEGER ioccur
```

cvar {input}

The string variable that Screen Formatting generates automatically in the form definition record. The form definition record defines the variable. If you do not want to use the automatically generated variable, include the following type statement (n is the number of characters in the variable):

```
CHARACTER*n cvar
```

ivstat {output}

The condition that gives you the status of the variable. The following values are possible:

Value	Meaning
0	No error occurred on the variable.
1	The program supplied a variable that does not match the strings defined for the variable.

Include the following type statement:

INTEGER ivstat

istat {output}

The variable that indicates the results of the subroutine. The following values are possible:

Value	Meaning
0	Routine completed successfully.
7	No space is available.
9	Form indentifier is invalid.
11	Variable name is unknown.
36	System error exists.
38	Variable name is invalid.
39	Form is pushed.
91	Occurrence is unknown.
145	Data value is bad.
147	Variable type is wrong.

Include the following type statement:

INTEGER istat

Remarks

- When the program calls either the FDREAD or FDSHOW subroutine, Screen Formatting replaces the string variable on the terminal screen.
- Before you replace a string variable, you must open the form on which it is replaced.
- You cannot replace a string variable for a pushed form.
- If the string variable is not valid, it is not replaced.
- If the form specifies that the data must be in upper case, Screen Formatting converts it to upper case before storing the data in the form.

Resetting a Form

Purpose FDRESF resets the form to the state specified by the form definition.

Format **CALL FDRESF (iform, istat)**

Parameters **iform** {input}

The identifier established when the form was opened. Include the following type statement:

INTEGER iform

istat {output}

The variable that indicates the results of the subroutine. The following values are possible:

Value	Meaning
0	Routine completed successfully.
7	No space is available.
9	Form indentifier is invalid.
36	System error exists.
39	Form is pushed.
145	Data value is bad.

Include the following type statement:

INTEGER istat

Remarks

- When the program calls either the FDREAD or FDSHOW subroutine, Screen Formatting displays the form on the terminal screen with the reset specifications.
- All variables belonging to the form have their initial values and display attributes. The form is in its defined position.
- Before you reset a form, you must open it.
- You cannot reset a pushed form.

Resetting an Object Attribute

Purpose

FDRESO resets the display attributes for an object to those specified in the form definition.

Format

CALL FDRESO (iform, oname, ioccur, istat)

Parameters

iform {input}

The identifier established when the form was opened. Include the following type statement:

INTEGER iform

oname {input}

The name of the object whose attributes are reset. Include the following type statement:

CHARACTER*31 oname

ioccur {input}

The occurrence of the object. For the first or only occurrence, use 1. Include the following type statement:

INTEGER ioccur

istat {output}

The variable that indicates the results of the subroutine. The following values are possible:

Value	Meaning
0	Routine completed successfully
7	No space is available.
9	Form indentifer is invalid.
20	Occurrence is invalid.
25	Object name is invalid.
33	Object name is unknown.
39	Form is pushed.
54	Form is not scheduled.
145	Data value is bad.

Include the following type statement:

INTEGER istat

Remarks

- You can reset the attributes of objects that are variable text, constant text, lines, or boxes.
- Before you reset the attribute of an object, you must open and either add or combine the form the object is on.
- When the program calls either the FDREAD or FDSHOW subroutine, Screen Formatting displays the object using the reset attributes.

Setting the Cursor Position

Purpose FDSETC sets the cursor to a selected position for later display.

Format **CALL FDSETC (iform, oname, ioccur, icp, istat)**

Parameters iform {input}

The identifier established when the form was opened. Include the following type statement:

```
INTEGER iform
```

oname {input}

The name of the object on which you want to set the cursor. Include the following type statement:

```
CHARACTER*31 oname
```

ioccur {input}

The integer specifying the occurrence of the object name. For the first occurrence, use 1. Include the following type statement:

```
INTEGER ioccur
```

icp {input}

The character position to which you want to set the cursor. For the first character position, use 1. Include the following type statement:

```
INTEGER icp
```

istat {output}

The variable that indicates the results of the subroutine. The following values are possible:

Value	Meaning
0	Routine completed successfully.
7	No space is available.
9	Form indentifer is invalid.
21	Character position is invalid.
25	Object name is invalid.
33	Object name is unknown.
36	System error exists.
39	Form is pushed.
54	Form is not scheduled.
86	Attribute name is unknown.
91	Occurrence is unknown.
134	No object variable is defined.
145	Data value is bad.

Include the following type statement:

```
INTEGER istat
```

Remarks

- Use this subroutine to alter the default sequence of the application user's entry of variables. In the default sequence, Screen Formatting places the cursor on the first input variable of the highest priority form. The highest priority form is the form last added, combined, or positioned.

  At terminals with protected fields, the user tabs from one variable text object to the next. The cursor starts at the top line of the form. It moves from left to right on each line. When no variable text object appears on a line, the cursor moves down to the next line. At terminals without protected fields, the user must move the cursor using the arrow keys or the tab and return keys.

- When you call either the FDREAD or FDSHOW subroutine, Screen Formatting updates the terminal screen with the cursor at the specified position.

- If the position you specify is not visible on the screen, Screen Formatting shifts the data to make the cursor visible.

- The cursor position is in effect only for the next screen update from reading or showing forms.

- Before you set the cursor position on a form, you must open the form and either add or combine it.

- You cannot set the cursor position in a pushed form.

Setting Line Mode

Purpose FDSETL begins line-by-line interaction with an application user.

Format **CALL FDSETL (istat)**

Parameters istat {output}

The variable that indicates the results of the subroutine. The following values are possible:

Value	Meaning
0	Routine completed successfully.
145	Data value is bad.

Include the following type statement:

INTEGER istat

Remarks

- Use this call for extended dialogues in line mode. For short dialogues, Screen Formatting automatically switches to the proper mode (line or screen), but resources used for screen mode interaction remain.
- This call releases all screen mode resources:
  - Open forms are closed.
  - The mode is set to line.

Setting an Object Attribute

Purpose FDSETO changes a display attribute for an object.

Format **CALL FDSETO (iform, oname, ioccur, aname, istat)**

Parameters **iform** {input}

The identifier established when the form was opened. Include the following type statement:

INTEGER iform

oname {input}

The name of the object whose display attribute is being set. Include the following type statement:

CHARACTER*31 oname

ioccur {input}

The occurrence of the object. For the first or only occurrence, use 1. Include the following type statement:

INTEGER ioccur

aname {input}

The name given to the display attribute when it was defined on the form. The attribute used here is defiined for the form and not for a specific object. When using Screen Design Facility, screen attributes are defined through the ATTRIB function. When using a CYBIL program, the ADD_DISPLAY_DEFINITION attribute record defines form attributes.

Include the following type statement:

CHARACTER*31 aname

istat {output}

The variable that indicates the results of the subroutine. The following values are possible:

Value	Meaning
0	Routine completed successfully.
7	No space is available.
9	Form indentifer is invalid.
20	Occurrence is invalid.
25	Object name is invalid.
29	Attribute name is invalid.
33	Object name is invalid.
39	Form is pushed.
54	Form is not scheduled.
86	Attribute name is unknown.
91	Occurrence is unknown.
145	Data value is bad.

Include the following type statement:

```
INTEGER istat
```

Remarks

- You can set the attributes of objects that are variable text, constant text, lines, or boxes.
- Changed attributes replace existing attributes.
- When you call either the FDREAD or FDSHOW subroutine, Screen Formatting displays the object using the set attributes.
- If the object you specify is not visible on the screen, Screen Formatting shifts the data to make the object visible.
- Before you set the attribute of an object, you must open the form the object is on and either add or combine it.
- You cannot set attributes of objects on a pushed form.

Showing Forms

Purpose FDSHOW updates the terminal screen.

Format **CALL FDSHOW (istat)**

Parameters istat {output}

A status variable that indicates the results of the subroutine. The following values are possible:

Value	Meaning
0	Routine completed successfully.
1	Terminal is disconnected.
7	No space is available.
36	System error exists.
53	No forms are scheduled for display.
131	Form is too large for screen.
145	Data value is bad.

Include the following type statement:

```
INTEGER istat
```

Remarks

- When none of the forms scheduled for display has an event or input variable defined, use this subroutine instead of FDREAD.
- When you do not want any input from the terminal user, use this subroutine.
- A call to FDSHOW:
  - Displays all the forms you have scheduled for display and have not deleted. If you added or combined forms since the last FDREAD or FDSHOW call, it displays them for the first time.
  - Removes from the screen the forms you deleted since the last FDREAD or FDSHOW call.
  - Displays variables replaced since last FDREAD or FDSHOW call.
  - Displays objects with attributes set or reset since last FDREAD or FDSHOW call.

Using Pascal to Manage Forms 5

Writing a Program to Use Forms 5-2
  Copying Procedure Definitions 5-2
    Pascal Procedures 5-2
    STATUS Parameter 5-3
    VARIABLE_STATUS Parameter 5-3
  Including Data Definitions 5-4
    Data Types for Forms 5-4
    Data Definitions for Parameters 5-5
    Special String Convention 5-5
  Calling Screen Formatting 5-6
    Displaying and Removing Forms and Variable Data 5-6
    Processing Events and Data 5-8
      Processing Normal Events 5-8
      Processing Abnormal Events 5-9
  Running a Prototype of the Application 5-9
  Example Program for Managing Forms with Pascal 5-11
    Forms Managed in the Program 5-11
    Design Specification 5-14
    Form Definition Decks 5-16
    Example Pascal Program 5-17

Expanding and Compiling a Program 5-27

Helping the User Start the Application 5-29
  Creating a User Procedure 5-29
  Creating a User Prolog 5-30
  Selecting a Natural Language 5-31
  Starting the Application 5-31

Pascal Procedure Calls for Interacting with Forms 5-32
  Adding a Form 5-33
  Changing Table Size 5-34
  Closing a Form 5-36
  Combining Forms 5-37
  Deleting a Form 5-39
  Getting an Integer Variable 5-40
  Getting the Next Event 5-42
  Getting a Real Variable 5-46
  Getting a String Variable 5-49
  Opening a Form 5-51
  Popping a Form 5-53
  Positioning a Form 5-54
  Pushing a Form 5-56
  Reading a Form 5-57

Replacing an Integer Variable 5-59
Replacing a Real Variable 5-61
Replacing a String Variable 5-63
Resetting a Form 5-65
Resetting an Object Attribute 5-66
Setting the Cursor Position 5-68
Setting Line Mode 5-70
Setting an Object Attribute 5-71
Showing Forms 5-73

Using Pascal to Manage Forms 5

Chapter 1 presented an example of creating and managing forms. It demonstrated that both the designer and the programmer have specific tasks to accomplish. When creating and managing forms using a Pascal program, the following tasks must be completed:

1. The form designer and programmer plan the forms and program.

2. The form designer creates the forms specifying Pascal as the form processor (or programming language) and prepares a design specification.

3. The form designer puts the forms in an object library and makes the form record definition available. Each record definition contains the VAR definitions of all variables defined on a particular form and is written in Pascal.

4. The programmer codes the program, including calls to Screen Formatting Pascal procedures based on the design specification. These calls manage the forms created by the designer.

5. The programmer expands and compiles the program.

6. The programmer writes a user procedure to start the application and helps the user set up the correct terminal environment for using the forms.

When these tasks are complete, the program and forms are ready for the application user.

Chapter 5 describes the tasks performed by the programmer and shows them executed in a Pascal program. At the end of the chapter, you will find format and parameter descriptions for each call to Pascal procedures used by Screen Formatting.

The designer's tasks and, also, the formats of the CYBIL procedure calls that create forms are described in chapter 7. (For information about designing forms using the Screen Design Facility, see the NOS/VE Screen Design Facility manual.)

Writing a Program to Use Forms

When writing a program to use forms, you must:

- Copy the procedure definitions for the Pascal procedures used by Screen Formatting.
- Include the data definitions generated by Screen Formatting when the designer creates the form. The data definitions hold values transferred to and from the form for the variable text objects.
- Call Screen Formatting procedures to manage the forms and the variable text objects on the forms.

Following the descriptions of these tasks is a Pascal program in which they're executed.

Copying Procedure Definitions

The procedure definitions describe the procedures and their parameters. Screen Formatting uses special definitions for:

- Each Pascal procedure.
- The STATUS parameter used on every procedure.
- The VARIABLE_STATUS parameter used on procedures that get and replace variables.

To use these definitions, copy the decks containing them into your program using the Source Code Utility *COPYC directives.

Pascal Procedures

To obtain the definitions for each procedure, copy FDP$PASCAL_PROCEDURES deck into your program. The following example shows the first procedure in the deck.

```
PROCEDURE fdp$xadd_form
   (VAR form_identifier: integer;
   VAR status: integer);
   CYBIL_EXTERNAL;
```

Although the procedures in FDP$PASCAL_PROCEDURES do not define strings using VAR, you should define string variables for parameters using VAR. How you define the variables for each procedure parameter is described in the call to the procedure later in this chapter.

STATUS Parameter

To obtain the values for the Pascal procedure status parameter, copy the FDE$PASCAL_PROCEDURE_STATUS deck into your program. The following example shows some of the contents of this deck:

```
CONST
   fde$call_successful = 0;
   fde$terminal_disconnected = 1;
   fde$no_input_requested = 2;
   fde$cursor_not_in_variable = 3;
   fde$more_errors_exist = 4;
```

Appendix E has a complete list of the contents of the deck.

VARIABLE_STATUS Parameter

To obtain the values for the VARIABLE_STATUS parameter, copy the FDT$VARIABLE_STATUS deck into your program. Following is the contents of this deck:

```
TYPE
   fdt$variable_status =
      (fdc$no_error,
      fdc$invalid_string,
      fdc$invalid_real,
      fdc$invalid_integer,
      fdc$unknown_user_value,
      fdc$invalid_bpd_data,
      fdc$no_digits,
      fdc$loss_of_significance,
      fdc$variable_not_filled,
      fdc$overflow,
      fdc$underflow,
      fdc$indefinite,
      fdc$infinite,
      fdc$variable_not_entered,
      fdc$output_format_bad,
      fdc$variable_truncated);
```

Use the ORD function in your program to check the variable status.

Including Data Definitions

In your program, you must include definitions of the data types of variables that are transferred to and from forms and definitions and those that are transferred for variables used by the parameters of the procedures called by Screen Formatting. There is also a special convention to follow when defining Pascal string variables used in Screen Formatting calls.

Data Types for Forms

The data type for each form resides on a *form definition record* created by the form designer. You transfer data in your program to and from variable text objects through this record.

When the designer creates a form, Screen Formatting generates a common deck that defines the form definition record. For example, Screen Formatting generated the following source file for a form named PASCAL_SELECT_FORM. (The form definition record name is the same as the form name.)

```
*DECK PASCAL_SELECT_FORM expand = false
  TYPE
    pascal_select_form = record
      object: string (1);
      message: string (40);
    end;
```

For this example, Screen Formatting was accessed through a CYBIL program. To use the Screen Design Facility, specify CYBIL as the language processor when creating the form. Then create the form definition record using a CYBIL program from the Examples online manual. See Creating Form Definition Records for Existing Forms in chapter 7.

The designer saves each form definition record as a deck on a NOS/VE source library using SOURCE_CODE_UTILITY (SCU).[1] The DECK directive in the file creates the correct name for the deck when it is processed.

---

1. Because each form has its own definition and the STATUS parameters use common decks, we recommend that you manage the source text using SCU. (For information on SCU, see the NOS/VE Source Code Management manual.)

At the beginning of your program, you must copy the form definition deck for each form created by the designer:

- Get the name of the deck from the design specification (the designer assigns the name while creating the form).
- Copy the deck by specifying its name on the SCU *COPY directive.

You then define storage to hold the variables for each form using the type defined in the form definition record. For example, if you copied the deck shown previously to define a TYPE of PASCAL_SELECT_FORM, then you would include the following VAR statement to define storage for the form:

```
VAR selection: pascal_select_form
```

Data Definitions for Parameters

You must also define storage for the variables used in the procedures. To accomplish this, code a variable declaration (VAR) for each variable used as a parameter on a Screen Formatting call. All parameters must be defined using VAR. The descriptions of the procedure calls later in this chapter show the syntax for the VAR statements you need to add.

Special String Convention

Because the Pascal procedures actually call CYBIL procedures, you need to be aware of a difference between Pascal and CYBIL in the definition of strings. A Pascal string has two lengths, the maximum length and the current length. Pascal assignment statements set the current length. CYBIL procedures use only the maximum length of a string. When CYBIL assigns a value to a string, the length of the string does not change as it does with Pascal. Therefore, to ensure that string data transfers properly, your Pascal assignment statements must set the current length equal to the maximum length.

The following example sets the length of the variable STRING_VARIABLE equal to its maximum length:

```
string_variable:= '';
   FOR n:=2 TO MAXLENGTH(string_variable) DO
      string_variable:=string_variable[1..n-1] + '';
```

Calling Screen Formatting

When writing a program that uses forms, you perform two basic tasks with Screen Formatting procedures:

- Displaying and removing forms and variable data on the application user's screen.
- Processing events executed by the user.

Displaying and Removing Forms and Variable Data

To control the display of forms and variable data on the user's screen, you perform the following steps in the given sequence:

1. Open the form.

   When you open a form, Screen Formatting locates it and allocates resources for processing the Screen Formatting calls that use the form.

   You need open a form only once, no matter how many times you use or update it. For this reason, begin an application program by opening all the forms you will use. However, when a form requires a large amount of storage for variables, you may want to open the form only when the application user needs it.

   (For the format of the call that opens forms, see *Opening a Form* later in this chapter).

2. Add the form.

   When you add a form, Screen Formatting schedules it for display on the application user's screen.

   To display more than one form at a time, add all the forms before you display them. Screen Formatting maintains a list of all forms you add. The last form you add to the list becomes the top form on the screen. Because forms are opaque, the top form covers other forms appearing in the same area.

   When the terminal user completes data entry, the cursor position indicates what form Screen Formatting should process. Variables on this form (and any forms combined with this one) are validated and updated. Variables on other forms are not updated or validated.

   (For the formats of the calls that schedule forms for display, see *Adding a Form* and *Combining Forms* later in this chapter.)

3. Read the form.

   When you read a form, Screen Formatting displays all the forms you've added.

   When a form has an event or input variable defined, reading forms also accepts data from the application user and displays values returned by the program.

   (For the format of the call that reads forms, see *Reading Forms* later in this chapter. When none of the forms scheduled for display has an event or input variable defined, you can use a similar call described in *Showing Forms* later in this chapter.)

4. Delete the form.

   When you delete a form, Screen Formatting deletes it from the list of forms scheduled for display. The next time you read a form, the deleted form is removed from the screen. However, the form remains available for use later in the program (you must reschedule it for display).

   (For the format of the call that deletes a form, see *Deleting a Form* later in this chapter.)

5. Close the form.

   When you close a form, Screen Formatting releases the resources the form uses, and the form is no longer available to the user or your program.

   (For the format of the call that closes a form, see *Closing a Form* later in this chapter.)

Processing Events and Data

When creating a form, the designer defines two types of events a user can execute to return control to the program: normal and abnormal.

- For normal events, the program performs requested actions such as getting variables, doing computations, and updating the form.
- For abnormal events, the program does such things as delete the form and go on, or stop using Screen Formatting.

Processing Normal Events

To process a normal event:

1. Get the name of the event and the position of the cursor from Screen Formatting.

   Screen Formatting validates the data the user enters (the form designer defined the validation rules) and transfers values of screen variables to its storage. The form designer may also have created error forms to be displayed when the user enters an incorrect value or presses a key not defined as an event.

   (For the format of the call that gets the event name and cursor position, see *Getting the Next Event* at the end of this chapter.)

2. Get the data from Screen Formatting storage and transfer it to program storage.

   (For formats of the calls that get data, see the following sections later in this chapter: *Getting an Integer Variable, Getting a Real Variable*, and *Getting a String Variable.*)

3. Replace the data in Screen Formatting storage with the data in program storage.

   (For formats of the calls that replace variables, see the following sections later in this chapter: *Replacing an Integer Variable, Replacing a Real Variable*, and *Replacing a String Variable.*)

You can also reset the variables on a form to their original state. (For formats of the calls that reset variables to their original state, see *Resetting a Form* and *Resetting an Object Attribute* later in this chapter.)

Processing Abnormal Events

To process an abnormal event:

1. Get the name of the event and the position of the cursor from Screen Formatting.

   Unlike a normal event, Screen Formatting neither validates user entries nor transfers values of screen variables to Screen Formatting storage.

   (For the format of the call that gets the event name and cursor position, see *Getting the Next Event* later in this chapter.)

2. Write your own procedure to perform the task the design specification assigns to the event. Typical actions for an abnormal event include:

   - Resetting a form and redisplaying it.
   - Moving the user to a new form for additional processing.
   - Returning the user to a previous form.
   - Stopping the program.

   The user's screen is updated upon reading the forms again or ending the program.

Running a Prototype of the Application

Once the forms have been created for your application, you can interactively run a prototype using the MANAGE_FORMS utility. This allows you to test the order in which the forms appear, and to interact with the forms as the application user will do.

An example of an application prototype is given in chapter 2 under the section named The Application Prototype. This prototype uses forms that were created specifically for use in an SCL procedure. To learn about using a prototype, you can run the prototype as described in the section.

Once you are familiar with the utility, you can also run a prototype using forms created for a Pascal program. Because the naming conventions for Pascal do not conflict with SCL naming conventions, the variables defined for the form can be used in the prototype without any conversion taking place.

To use the prototype with the Pascal forms that are on the library specified in the prototype example, rather than opening the SCL forms listed, open the forms named:

PASCAL_SELECT_FORM
PASCAL_RECTANGLE_FORM
PASCAL_CIRCLE_FORM

One variable of type RECORD is created for each form as described for the SCL forms shown in the prototype example. The variable has the same name as the form. You can display the data structure of each variable using the DISPLAY_VALUE command. For example, to display the data structure for the PASCAL_RECTANGLE_FORM variable, enter the following command:

```
mf/display_value value=pascal_rectangle_form ..
mf../display_options=data_structure
display option: DATA_STRUCTURE

"RECORD"
  SIDE_TABLE: "ARRAY"
               1. "RECORD"
                   SIDE: "INTEGER"  0
                  "RECORD END"

               2. "RECORD"
                   SIDE: "INTEGER"  0
                  "RECORD END"
             "ARRAY END"

  AREA: "INTEGER"  0
  MESSAGE: "STRING"  '                                          '
"RECORD END"
```

For more information about the structure of variables that are records, see the NOS/VE System Usage manual.

Example Program for Managing Forms with Pascal

The program in this example computes the area of circles and rectangles. The example includes:

- Pictures of the forms managed in the program.
- The design specification supplied by the form designer.
- The form definition decks.
- The example program.

Forms Managed in the Program

The example program manages three forms residing on an object library named EXAMPLE_OBJECT_LIBRARY that must be in the user's command list.

When a user starts the application, Select Form appears (figure 5-1).

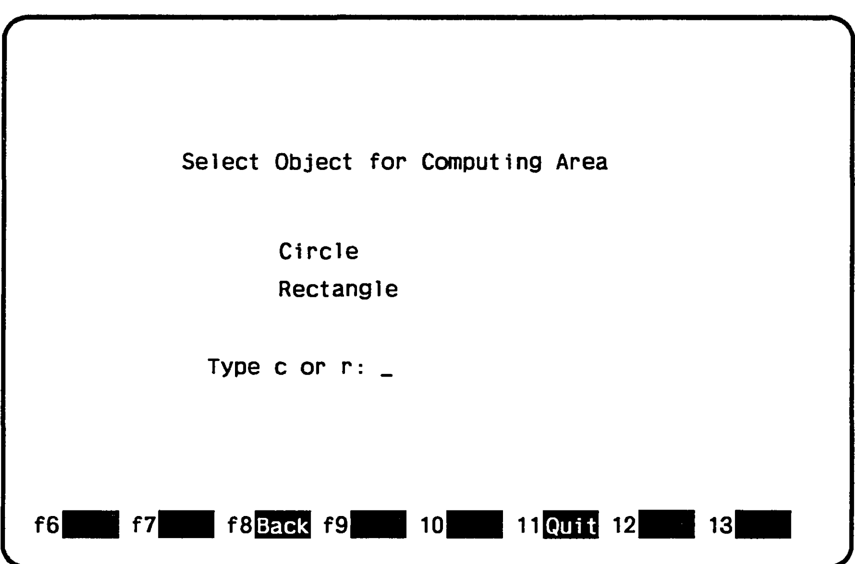


Figure 5-1. Select Form

On Select Form, a user enters either *c* to compute the area of a circle or *r* to compute the area of a rectangle.

When a user enters *r* on Select Form, Rectangle Form (figure 5-2) appears.

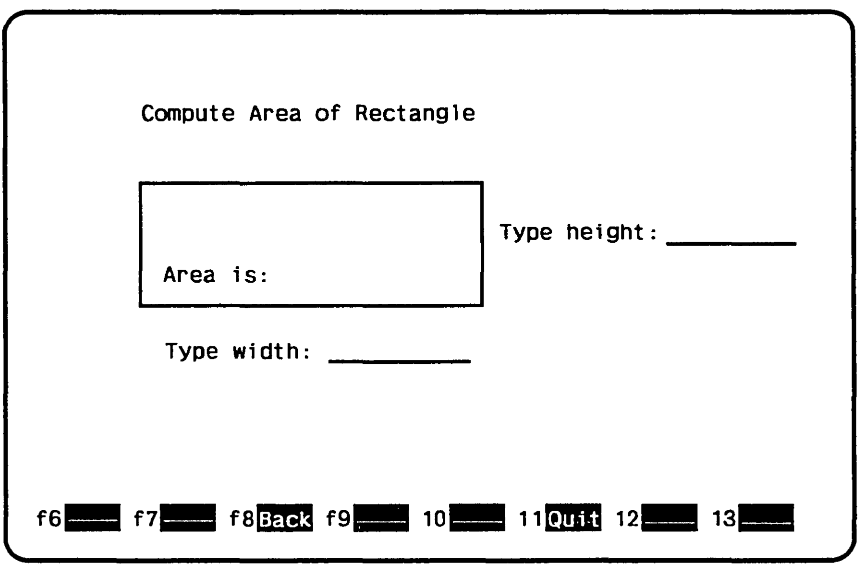


Figure 5-2. Rectangle Form

On Rectangle Form, the user enters the lengths of the sides of the rectangle as integers and presses the return key to have the program compute the area.

When a user enters *c* on Select Form, Circle Form (figure 5-3) appears.

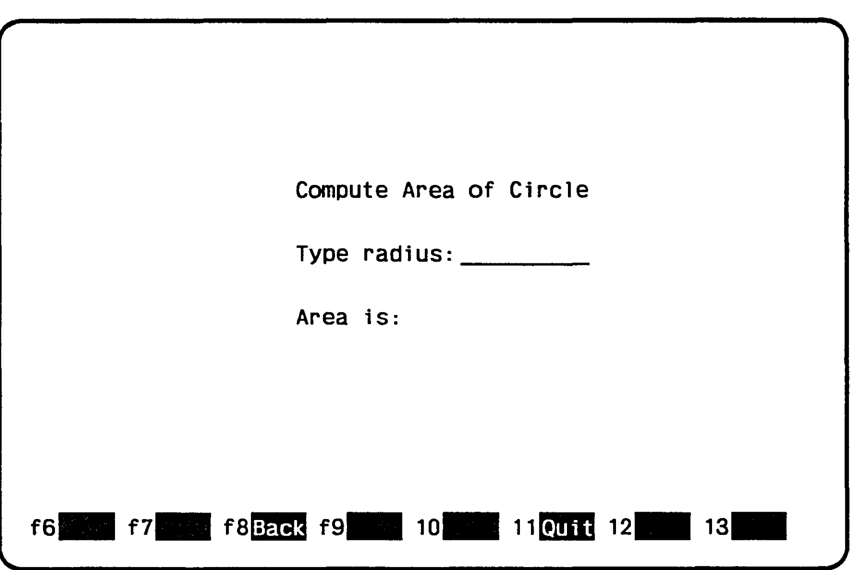


Figure 5-3. Circle Form

On Circle Form, the user enters the radius of the circle as a real value and presses the return key to have the program compute the area.

Design Specification

In writing the example program, the programmer uses the information listed in the following design specification:

- The names for the three forms used by the program are:

  PASCAL_SELECT_FORM
  PASCAL_RECTANGLE_FORM
  PASCAL_CIRCLE_FORM

- The user can call both the Rectangle Form and Circle Form from the Select Form.

- The following variable text objects are defined on the forms:

Variable Object	Description
Select Form:	
MESSAGE	Area for displaying error messages.
OBJECT	Area for user input of *r* or *c*.
Rectangle Form:	
SIDE_TABLE	Table that holds values for the rectangle's sides.
SIDE	Areas (two) for user input of values for the rectangle's sides.
AREA	Area for returning value of computed area.
MESSAGE	Area for displaying error messages.
Circle Form:	
RADIUS	Area for user input of value for the circle's radius.
AREA	Area for returning value of computed area.
MESSAGE	Area for displaying error messages.

- The following events are defined on the forms:

Event	Description
COMPUTE	A normal program event that processes data entered on the form by the user. For Select Form, the COMPUTE event checks whether the user entered *r* or *c* and then displays the appropriate form. For the other forms, COMPUTE calculates the area and redisplays the form.
BACK	An abnormal program event that takes the user back to a previous environment. For Select Form, the BACK event stops the program. For the other forms, BACK returns the user to Select Form.
QUIT	An abnormal program event that stops the program.

Form Definition Decks

When the designer creates the three forms (by writing a program or by using Screen Design Facility), a form definition record is created along with each form. For the example program, the programmer copies the following form definition decks placed on an SCU library by the designer. The library in this example is named EXAMPLE_SOURCE_LIBRARY.

The PASCAL_SELECT_FORM deck:

```
TYPE
  pascal_select_form = record
    object: string (1);
    message: string (40);
  end;
```

The PASCAL_RECTANGLE_FORM deck:

```
TYPE
  pascal_rectangle_form = record
    side_table: array [1 .. 2] of record
      side: integer
    end;
    area: integer;
    message: string (40);
  end;
```

The PASCAL_CIRCLE_FORM deck:

```
TYPE
  pascal_circle_form = record
    area: real;
    radius: real;
    message: string (40);
  end;
```

Example Pascal Program

This Pascal program calls the forms and executes the events described in the previous sections. The program is in the SCU deck named PASCAL_COMPUTE_OBJECT_AREA. To run the example program, see the Screen Formatting examples in the Examples online manual.

```
PROGRAM pascal_compute_object_area (output);

*copyc fdp$pascal_procedures
*copyc fde$pascal_procedure_status
*copyc fdt$variable_status
*copyc pascal_select_form

VAR

{ Variables for select form.}

  select_form_identifier: integer;
  selection: pascal_select_form;

  character_position: integer;
  circle_form_identifier: integer;
  event_name: string(31);
  event_normal: string(1);
  form_identifier: integer;
  event_type: integer;
  form_name: string (31);
  form_x_position: integer;
  form_y_position: integer;
  integer_variable: integer;
  last_event: string (1);
  object_name: string(31);
  object_occurrence: integer;
  object_type: integer;
  object_x_position: integer;
  object_y_position: integer;
  occurrence: integer;
  rectangle_form_identifier: integer;
  screen_x_position: integer;
  screen_y_position: integer;
  status: integer;
  status_message: string (60);
  variable_name: string(31);
  variable_status: integer;
```

```
LABEL 20, 30;

PROCEDURE check_status
  (message: PACKED ARRAY [lower .. upper: integer] OF CHAR;
  status: integer);

  VAR
    n: integer;
    p_string: ^string;

  BEGIN
   IF status <> fde$call_successful THEN
     BEGIN
       NEW (p_string, upper);
       p_string^ := message [1];
       FOR n := lower + 1 TO upper DO
         p_string^ := p_string^ [1 .. n-1] + message [n];
       writeln (p_string^, ', status = ', status);
       GOTO 30
     END;
  END;

  PROCEDURE display_variable_status
      (form_message: PACKED ARRAY [lower .. upper: integer]
       of CHAR;
       VAR form_identifier: integer);

    VAR
      n: integer;
      message: string(40);

      BEGIN
        message := form_message [1];
        FOR n := lower + 1 TO upper DO
          message := message [1 .. n-1] + form_message [n];
        FOR n := upper TO 39 DO
          message := message [1 .. n] + ' ';
```

```
        occurrence := 1;
        variable_name := 'MESSAGE                          ';
        fdp$xreplace_string_variable (form_identifier,
              variable_name,occurrence, message,
              variable_status, status);
        check_status (' Replace string failed on message.',
              status)
  END;

PROCEDURE compute_circle_area;

{ Variables for circle form.}

*copy pascal_circle_form

VAR
  circle: pascal_circle_form;

  LABEL 5, 10;
  BEGIN
    fdp$xreset_form (circle_form_identifier, status);
    check_status (' Reset failed on form circle', status);
    fdp$xadd_form(circle_form_identifier, status);
    check_status (' Add failed on form circle', status);

 5: REPEAT
      fdp$xread_forms (status);
      check_status (' Read failed on form circle', status);

      fdp$xget_next_event (event_name, event_normal,
            screen_x_position,screen_y_position,
            form_identifier, form_x_position, form_y_position,
            event_type, object_name, object_occurrence,
            character_position, object_type, object_x_position,
            object_y_position, last_event, status);
      check_status (' Get next event failed on form circle',
            status);
```

```
        IF event_name <> 'COMPUTE' THEN
          BEGIN
            FDP$XDELETE_FORM (circle_form_identifier, status);
            check_status (' Delete failed on form circle', status);
            GOTO 10
          END;

        variable_name := 'RADIUS                         ';
        occurrence := 1;
        fdp$xget_real_variable (circle_form_identifier,
            variable_name, occurrence, circle.radius,
            variable_status, status);
        check_status (' Get failed on form circle', status);
        IF variable_status <> ORD(fdç$no_error) THEN
          BEGIN
            display_variable_status
                ('Type valid value for radius.',
                circle_form_identifier);
             GOTO 5
          END;

       circle.area := 3.14 * (circle.radius * circle.radius);
       variable_name := 'AREA                            ';
       fdp$xreplace_real_variable (circle_form_identifier,
             variable_name, occurrence, circle.area,
             variable_status, status);
      check_status (' Replace real failed on form circle', status);
      IF variable_status <> ORD(fdc$no_error) THEN
        BEGIN
          display_variable_status ('Format cannot display area.',
              circle_form_identifier);
          GOTO 5
        END;
       display_variable_status (' ', circle_form_identifier);
      UNTIL FALSE;

  10: END;
```

```
PROCEDURE compute_rectangle_area;

{ Variables for rectangle form.}

*copy pascal_rectangle_form

VAR
  display_name: string  (31);
  rectangle: pascal_rectangle_form;

  LABEL 5, 10;
  BEGIN
    display_name := 'ERROR                          ';
    fdp$xreset_form (rectangle_form_identifier, status);
    check_status (' Reset failed on form rectangle', status);
    fdp$xadd_form(rectangle_form_identifier, status);
    check_status (' Add failed on form rectangle', status);

 5: REPEAT
      fdp$xread_forms (status);
      check_status (' Read failed on form rectangle', status);

      fdp$xget_next_event (event_name, event_normal,
            screen_x_position, screen_y_position,
            form_identifier, form_x_position, form_y_position,
            event_type, object_name, object_occurrence,
            character_position, object_type, object_x_position,
            object_y_position, last_event, status);
      check_status (' Get next event failed on form rectangle',
            status);

      IF event_name <> 'COMPUTE' THEN
        BEGIN
          FDP$XDELETE_FORM (rectangle_form_identifier, status);
          check_status (' Delete failed on form rectangle',
                status);
          GOTO 10
        END;
```

```
{ Remove any previous error indications.}

      variable_name := 'SIDE                           ';
      occurrence := 1;
      fdp$xreset_object_attribute (rectangle_form_identifier,
           variable_name, occurrence, status);
      occurrence := 2;
      fdp$xreset_object_attribute (rectangle_form_identifier,
           variable_name, occurrence, status);
      display_variable_status (' ', rectangle_form_identifier);

{ Get values terminal user entered for rectangle sides.}

      variable_name := 'SIDE                           ';
      fdp$xget_integer_variable (rectangle_form_identifier,
          variable_name, occurrence,
          rectangle.side_table[1].side,
          variable_status, status);
       check_status (' Get failed on form rectangle', status);
       IF variable_status <> ORD(fdc$no_error) THEN
        BEGIN
          character_position := 1;
          fdp$xset_cursor_position (rectangle_form_identifier,
              variable_name, occurrence, character_position,
              status);
          fdp$xset_object_attribute (rectangle_form_identifier,
              variable_name, occurrence, display_name, status);
          display_variable_status
              ('Type valid value for height.',
              rectangle_form_identifier);
           GOTO 5
        END;
```

```
      occurrence := 2;
      fdp$xget_integer_variable (rectangle_form_identifier,
          variable_name, occurrence,
          rectangle.side_table[2].side,
          variable_status, status);
      check_status (' Get failed on form rectangle', status);
      IF variable_status <> ORD(fdc$no_error) THEN
        BEGIN
          character_position := 1;
          fdp$xset_cursor_position (rectangle_form_identifier,
              variable_name, occurrence, character_position,
              status);
          fdp$xset_object_attribute (rectangle_form_identifier,
              variable_name, occurrence, display_name, status);
          display_variable_status ('Type valid value for width.',
              rectangle_form_identifier);
           GOTO 5
        END;

     rectangle.area := rectangle.side_table[1].side *
         rectangle.side_table [2].side;
     variable_name := 'AREA                          ';
     occurrence := 1;
     fdp$xreplace_integer_variable (rectangle_form_identifier,
         variable_name, occurrence, rectangle.area,
         variable_status, status);
    check_status (' Replace failed on form rectangle', status);
    IF variable_status <> ORD(fdc$no_error) THEN
      BEGIN
        display_variable_status ('Format cannot display area.',
            rectangle_form_identifier);
        GOTO 5
      END;

    UNTIL FALSE;

10: END;
```

```
BEGIN

{ Initialize variable values and string lengths.}

  event_normal := ' ';
  last_event := ' ';
  event_name     := '                              ';
  object_name    := '                              ';
  selection.object := ' ';

{ Open all forms.}

  form_name :=    'PASCAL_SELECT_FORM             ';
  fdp$xopen_form(form_name, select_form_identifier, status);
  check_status (' Open failed on form select', status);

  form_name :=    'PASCAL_CIRCLE_FORM             ';
  fdp$xopen_form(form_name, circle_form_identifier, status);
  check_status (' Open failed on form circle', status);

  form_name :=    'PASCAL_RECTANGLE_FORM        ';
  fdp$xopen_form(form_name, rectangle_form_identifier, status);
  check_status (' Open failed on form rectangle', status);

  fdp$xadd_form(select_form_identifier, status);
  check_status (' Add failed on form select', status);

{ Process terminal user input for select form.}

20: REPEAT
    fdp$xread_forms (status);
    check_status (' Read failed on form select', status);

    fdp$xget_next_event (event_name, event_normal,
          screen_x_position, screen_y_position, form_identifier,
          form_x_position, form_y_position,
          event_type, object_name, object_occurrence,
          character_position, object_type, object_x_position,
          object_y_position, last_event, status);
    check_status (' Get next event failed on form select',
          status);
```

```
      IF event_name <>  'COMPUTE' THEN
        GOTO 30;

    variable_name := 'OBJECT                          ';
    occurrence := 1;
    fdp$xget_string_variable (select_form_identifier,
        variable_name, occurrence, selection.object,
        variable_status, status);
    check_status (' Get failed on form select', status);
    IF variable_status <> ORD(fdc$no_error) THEN
      BEGIN
        display_variable_status
            ('Type valid value for selection.',
            select_form_identifier);
         GOTO 20
      END;

    IF selection.object = 'R' THEN

{ Compute area of rectangle. }

       BEGIN
         FDP$XDELETE_FORM (select_form_identifier, status);
         check_status (' Delete failed on form select', status);
         compute_rectangle_area;
       END

    ELSE
      IF selection.object = 'C' THEN

{ Compute area of circle. }

        BEGIN
          FDP$XDELETE_FORM (select_form_identifier, status);
          check_status (' Delete failed on form select', status);
          compute_circle_area
        END
    ELSE
```

```
{ Invalid selection. }

      BEGIN
        display_variable_status ('Type r or c.',
            select_form_identifier);
        GOTO 20;
      END;

  IF event_name = 'QUIT' THEN
    GOTO 30;

  FDP$XRESET_FORM (select_form_identifier, status);
  check_status (' Reset failed on form select', status);

  FDP$XADD_FORM (select_form_identifier, status);
  check_status (' Add failed on form select', status);

UNTIL FALSE;

{ Close all forms. }

30: FDP$XCLOSE_FORM (select_form_identifier, status);
    check_status (' Close failed on form select', status);

    FDP$XCLOSE_FORM (circle_form_identifier, status);
    check_status (' Close failed on form circle', status);

    FDP$XCLOSE_FORM (rectangle_form_identifier, status);
    check_status (' Close failed on form rectangle', status)
 END.
```

Expanding and Compiling a Program

Programs using Screen Formatting use common decks and form definition records that reside outside the main program. To manage the source text for this type of program, put the program in one or more SCU decks. This allows you to update individual parts of a program and to use forms in more than one program without duplicating code.[2]

To expand and compile a program maintained in SCU decks:

1. Expand the deck containing the main program.

2. Compile the expanded program.

3. Put the compiled program on an object library.

A procedure for compiling and expanding a program is shown in the following example. (The example is based on the example program and form definition records described earlier. The example shows how to place decks on library EXAMPLE_SOURCE_LIBRARY.)

The procedure calls SCU to expand the SCU directives contained in the program. For this expansion, you must include the $SYSTEM.CYBIL.OSF$PROGRAM_INTERFACE library as an alternate base. The program is then compiled and put on an object library.

```
PROCEDURE pascal_compile_deck, pascd (
  deck,  d: name = $required
  status)

  source_code_utility
    use_library base=example_source_library result=$null
    expand_deck deck=deck ..
    compile=$local.compile ..
    alternate_base=$system.cybil.osf$program_interface
  quit
```

2. For information on SCU, see the NOS/VE Source Code Management manual.

```
     pascal input=$local.compile ..
        list=$local.list runtime_checks=all  ..
       binary=$local.lgo  ..
       debug_aids=all

     create_object_library
       add_module library=example_object_library
       combine_module library=$local.lgo
       generate_library library=example_object_library.$next
     quit

  PROCEND pascal_compile_deck
```

To use the procedure, put it on library EXAMPLE_OBJECT_ LIBRARY and then add the library to your command list (using the CREATE_COMMAND_LIST_ENTRY command). You can execute the procedure by entering:

```
/pascal_compile_deck deck=pascal_compute_object_area
```

The compiled program is now also on library EXAMPLE_OBJECT_ LIBRARY.

For more information on writing and using procedures, see the NOS/VE System Usage manual.

Helping the User Start the Application

The complete application consists of your program and the forms created by the designer. To integrate the forms with your program, you must:

- Create a procedure that gives users access to the object library containing the forms and program.
- Ensure that the user's terminal environment is set up properly to use the forms (in most instances, by creating a user prolog).
- Ensure that users select the correct natural language.
- Ensure that users know how to start the application.

Creating a User Procedure

To give the user access to the object library containing the forms:

1. Write a NOS/VE procedure from which the user starts the application.
2. Place the procedure on the library that contains the compiled program.

For example, the following procedure executes the application that uses the starting procedure PASCAL_COMPUTE_OBJECT_AREA on library EXAMPLE_OBJECT_LIBRARY. The other libraries accessed by the program are $SYSTEM.FDF$LIBRARY and $SYSTEM.TDU.TERMINAL_DEFINITIONS. Users must have these libraries available in order for the program to call the Screen Formatting procedures.

```
PROCEDURE pascal_compute_area, pasca (
  status)

  execute_task ..
    library=(example_object_library,$system.fdf$library,..
    $system.tdu.terminal_definitions) ..
    starting_procedure=pascal_compute_object_area

PROCEND pascal_compute_area
```

Creating a User Prolog

To ensure that the users' terminal environment is set up properly to use the forms, the user must set the following terminal characteristics before he or she executes the procedure:

Characteristic	Description
Terminal model	Identifies the terminal to NOS/VE.
Attention character	Provides a character users can enter to interrupt the application.
Hold messages	Tells the network to hold all network messages until the user stops the application. Otherwise, a computer operator message may overwrite a form while a user is entering data, confusing the user.

In most instances, users should set up their terminal for the entire terminal session in their user prologs. The example below does the following:

- Identifies a Digital Equipment Corporation VT220 terminal to the system.
- Chooses the exclamation point as a way to interrupt the program.
- Holds all messages from a NAMVE/CDCNET network.
- Sets up the way the terminal uses the exclamation point to interrupt the program.

The users add the following commands to their user prologs:

```
change_terminal_attributes terminal_model=dec_vt220 ..
  attention_character='!' ..
  status_action=hold
change_term_conn_defaults attention_character_action=1
change_connection_attributes terminal_file_name=input aca=1
change_connection_attributes terminal_file_name=output aca=1
change_connection_attributes terminal_file_name=command aca=1
```

For a further explanation of how to interrupt a screen application during an interactive session, and what commands to use for networks other than NAMVE/CDCNET, see the NOS/VE System Usage manual.

Selecting a Natural Language

To ensure that users receive messages in the correct natural language, have them add the CHANGE_NATURAL_LANGUAGE command to their prologs. Because the default language is US_ENGLISH and all messages returned by Screen Formatting are in this language, have users include this command only when you have changed messages to another language.

Changing messages to other languages is described in the NOS/VE Object Code Management manual. The CHANGE_NATURAL_LANGUAGE command is described in the NOS/VE System Usage manual.

Starting the Application

To start the application, the users enter:

```
/create_command_list_entry e=example_object_library
/pascal_compute_area
```

When finished with the application, the users remove the object library from their command lists:

```
/delete_command_list_entry e=example_object_library
```

Pascal Procedure Calls for Interacting with Forms

The following sections describe the Pascal procedure calls to Screen Formatting modules. For each procedure, there is a purpose description, input format, list of parameters and their types, condition identifiers, and pertinent remarks.

An application program calls Screen Formatting procedures to interact with an application user through the use of forms. Each of these procedures is defined as a deck on the SCU library $SYSTEM.CYBIL.OSF$PROGRAM_INTERFACE. This library must be in the alternate base when compiling the application program.

These procedures are external routines that reside on the library called $SYSTEM.FDF$LIBRARY. This library must be in the user's program library list in order to execute the program.

For detailed information on Pascal procedure calls, see the Pascal for NOS/VE manual.

When checking the status of procedures, you must check the STATUS parameter and also check, when present, the VARIABLE_STATUS parameter. If the value of STATUS is zero, you can process output from the procedure. However, if there is also a VARIABLE_STATUS parameter, check its value before using variable output form the procedure. The variable status is independent of the status for the procedure.

Adding a Form

Purpose FDP$XADD_FORM schedules a form for display on the application user's screen.

Format **FDP$XADD_FORM (form_identifier, status)**

Parameters form_identifier {input}

The identifier established when the form was opened. Include the following variable declaration:

```
VAR form_identifier : integer;
```

status {output}

The variable that indicates the results of the procedure. Include the following variable declaration:

```
VAR status : integer;
```

Conditions

fde$bad_data_value
fde$form_already_added
fde$form_pushed
fde$form_too_large_for_screen
fde$invalid_form_identifier
fde$no_space_available
fde$system_error

Remarks

- When you call either the FDP$XREAD_FORMS or FDP$XSHOW_FORMS procedure, Screen Formatting displays the added form on the terminal screen. The added form is placed on top of other forms occupying the same area on the screen.

- When displayed, each form that is added operates independently from other forms that have been added. When a user executes a normal event, Screen Formatting validates and updates only those variables on the form associated with the event. To have forms share events, see Combining Forms later in this section.

- Before you add a form, you must open it.

- You cannot add a pushed form.

Changing Table Size

Purpose FDP$XCHANGE_TABLE_SIZE changes the size of the table during program execution.

Format **FDP$XCHANGE_TABLE_SIZE (form_identifier, table_name, table_size, status)**

Parameters form_identifier {input}

The identifier established when the form was opened. Include the following variable declaration:

```
VAR form_identifier : integer;
```

table_name {input}

The name of the table to change in size. Include the following variable declaration:

```
VAR table_name : string(31);
```

table_size {input}

The size of the table. While this procedure is in effect, Screen Formatting limits the number of stored occurrences allowed for a table to the value you specify on this parameter. How many occurrences are displayed at one time depends on the number of visible occurrences defined in the form.

If you specify zero for the table size, no occurrences appear on the form.

Include the following variable declaration:

```
VAR table_size : integer;
```

status {output}

The record that indicates the results of the procedure. Include the following variable declaration:

```
VAR status : integer;
```

Conditions fde$bad_data_value
fde$form_pushed
fde$invalid_form_identifier
fde$invalid_table_name
fde$invalid_table_size
fde$no_space_available

fde$unknown_table_name

Remarks

- The table must be present in an open form.
- The size limitation remains in effect until the next time you call the FDP$XCHANGE_TABLE_SIZE procedure.
- The maximum size for a table is identified by the form as the maximum number of stored occurrences. If you specify a table size larger than the maximum, you receive an error message (FDE$INVALID_TABLE_SIZE).

Examples

The following examples describe how changing the size of a table affects the application user. On the form, the table's specifications are a maximum of 20 stored occurrences, of which 6 occurrences can be visible at one time.

- If you specify a table size of 10, Screen Formatting displays 6 occurrences and allows the application user to page to the 10th occurrence.
- If you specify a table size of 4, Screen Formatting displays 4 occurrences and does not allow the application user to page.

Closing a Form

Purpose FDP$XCLOSE_FORM releases resources used to process a form and deletes the form from the list scheduled for display.

Format **FDP$XCLOSE_FORM (form_identifier, status)**

Parameters form_identifier {input}

The identifier established when the form was opened. Include the following variable declaration:

```
VAR form_identifier : integer;
```

status {output}

The record that indicates the results of the procedure. Include the following variable declaration:

```
VAR status : integer;
```

Conditions

fde$bad_data_value
fde$invalid_form_identifier
fde$form_pushed
fde$no_space_available

Remarks

- When the program calls either the FDP$XREAD_FORMS or FDP$XSHOW_FORMS procedure, Screen Formatting removes the closed form from the terminal screen as a result of calling this procedure.
- Before you can close a form, you must open it.
- You cannot close a pushed form.

Combining Forms

Purpose

FDP$XCOMBINE_FORM combines a form with a previously added form and schedules the combined form for display on the terminal screen.

Format

FDP$XCOMBINE_FORM (added_form_identifier, combine_form_identifier, status)

Parameters

added_form_identifier {input}

The identifier for this instance of the previously added form. Include the following variable declaration:

```
VAR form_identifier : integer;
```

combine_form_identifier {input}

The identifier for the form you are combining with the previously added form. Include the following variable declaration:

```
VAR form_identifier : integer;
```

status {output}

The record that indicates the results of the procedure. Include the following variable declaration:

```
VAR status : integer;
```

Conditions

fde$bad_data_value
fde$form_already_added
fde$form_already_combined
fde$form_pushed
fde$form_too_large_for_screen
fde$invalid_form_identifier
fde$no_space_available
fde$system_error

Remarks

- You cannot combine a pushed form.
- The combined form inherits the event definitions of the previously added form.
- Before you combine a form with a previously added form, you must open both forms.
- When the programs calls either the FDP$XREAD_FORMS or FDP$XSHOW_FORMS procedure, Screen Formatting displays the combined form. The combined form is placed on top of other forms occupying the same area on the screen.
- When the application user executes an event to return to the program normally, Screen Formatting updates all program variables associated with both the added and combined forms.
- To combine several forms with a previously added form, call this procedure more than once.

Deleting a Form

Purpose FDP$XDELETE_FORM deletes the form from the list of forms scheduled for display.

Format **FDP$XDELETE_FORM (form_identifier, status)**

Parameters **form_identifier** {input}

The identifier established when the form was opened. Include the following variable declaration:

```
VAR form_identifier : integer;
```

status {output}

The record that indicates the results of the procedure. Include the following variable declaration:

```
VAR status : integer;
```

Conditions

fde$bad_data_value
fde$form_not_scheduled
fde$form_pushed
fde$invalid_form_identifier
fde$no_space_available

Remarks

- When the program calls either the FDP$XREAD_FORMS or FDP$XSHOW_FORMS procedure, Screen Formatting removes the deleted form from the terminal screen and replots any forms uncovered by the deleted form.
- When you add a form (FDP$XADD_FORM) again that you previously deleted, the data in the form is retained.
- Before you delete a form, you must open it.
- You cannot delete a pushed form.
- If the form was added and has any combined forms associated with it, the combined forms are also deleted.
- When you delete a combined form, only that form is deleted. Areas covered by the combined form are replotted after the combined form is deleted.

Getting an Integer Variable

Purpose FDP$XGET_INTEGER_VARIABLE gets the value the user entered on the form for an integer variable and transfers it to the program.

Format **FDP$XGET_INTEGER_VARIABLE (form_identifier, name, occurrence, variable, variable_status, status)**

Parameters form_identifier {input}

The identifier established when the form was opened. Include the following variable declaration:

VAR form_identifier : integer;

name {input}

The name of the integer variable to get and transfer to the program. This name was defined when the form was created.

occurrence {input}

The occurrence of the variable name. The values allowed are 1 .. 1000. Use 1 for the first or only occurrence. Include the following variable declaration:

VAR occurrence : integer;

variable {output}

The integer variable that Screen Formatting generates automatically in the form definition record. If you do not want to use the automatically generated variable, use a variable of type integer.

variable_status {output}

An ordinal that gives you the status of the variable. The following values are possible:

FDC$INVALID_BDP_DATA

The user entered data that does not correspond to the defined program data type.

FDC$INVALID_INTEGER

The user entered data that is not in the range defined for the variable.

FDC$LOSS_OF_SIGNIFICANCE

The user entered an integer that is too large.

FDC$NO_ERROR

No error occurred on the variable.

Include the following variable declaration:

```
VAR variable_status : integer;
```

status {output}

The record that indicates the results of the procedure. Include the following variable declaration:

```
VAR status : integer;
```

Conditions

fde$bad_data_error
fde$invalid_form_identifier
fde$invalid_variable_name
fde$no_space_available
fde$system_error
fde$unknown_occurrence
fde$unknown_variable_name
fde$wrong_variable_type

Remarks

- Before you get an integer variable, you must open its form. If you get the variable after opening the form and before reading or replacing the variable on the form, the program returns the initial value specified by the form designer.

- If the form designer specifies data validation rules and error processing to display an error message or form, the program does not need to look at the variable status parameter.

  If the form designer specifies data validation rules and no error processing, the program must look at the variable status parameter.

  If the form designer specifies no data validation rules, the program must look at the variable status parameter.

Getting the Next Event

Purpose FDP$XGET_NEXT_EVENT gets the next event resulting from the most recent FDP$XREAD_FORMS procedure.

Format **FDP$XGET_NEXT_EVENT (event_name, event_normal, screen_x_position, screen_y_position, form_identifier, form_x_position, form_y_position, event_type, object_name, object_occurrence, character_position, object_type, object_x_position, object_y_position, last_event, status)**

Parameters event_name {output}

A data name to receive the application user's event. Include the following variable declaration:

```
VAR event_name : string(31);
```

event_normal {output}

A data name to receive the event normal indication. If the event is normal, T is returned; if the event is abnormal, F is returned. Include the following variable declaration:

```
VAR event_normal : string(1);
```

screen_x_position {output}

A data name to receive the screen x position of the event. The character position in the upper left corner of a screen is 1; the x position increases by 1 for each character, counting from left to right. Include the following variable declaration:

```
VAR screen_x_position : integer;
```

screen_y_position {output}

A data name to receive the screen y position of the event. The character position in the upper left corner of a screen is 1; the y position increases by 1 for each character counting from top to bottom. Include the following variable declaration:

```
VAR screen_y_position : integer;
```

form_identifier {output}

The identifier of the form on which the event occurred. Include the following variable declaration:

```
VAR form_identifier : integer;
```

form_x_position {output}

A data name to receive the form x position of the event. The character position in the upper left corner of a form is 1; the x position increases by 1 for each character, counting from left to right. Include the following variable declaration:

```
VAR form_x_position : integer;
```

form_y_position {output}

A data name to receive the form y position of the event. The character position in the upper left corner of a form is 1; the y position increases by 1 for each character counting from top to bottom. Include the following variable declaration:

```
VAR form_y_position : integer;
```

event_type {output}

A data name to receive the integer representing the type of the event. The following values can be returned:

Value	Meaning
0	The event occurred in a form, but not in an object.
1	The event occurred in an object.

Include the following variable declaration:

```
VAR event_type : integer;
```

object_name {output}

A data name to receive the object name where the event occurred. If the object doesn't have a name, the name is all spaces. This parameter is used only when the event occurs in an object. Include the following variable declaration:

```
VAR object_name : string(31);
```

object_occurrence {output}

A data name to receive the occurrence of the object. The first or only occurrence is returned as 1. This parameter is used only when the event occurs in an object. Include the following variable declaration:

```
VAR object_occurrence : integer;
```

character_position {output}

A data name to receive the character position within the object name where the event occurred. The first character position is 1. This parameter is used only when the event occurs in an object. Include the following variable declaration:

```
VAR character_position : integer;
```

object_type {output}

A data name to receive the type of object. This parameter is used only when the event occurs in an object. The following values can be returned:

Value	Meaning
0	The event occurred on a box object.
1	The event occurred on a constant text object.
2	The event occurred on a constant text box object.
3	The event occurred on a line object.
5	The event occurred on a variable text object.
6	The event occurred on a variable text box object.

Include the following variable declaration:

```
VAR object_type : integer;
```

object_x_position {output}

A data name to receive the x position of the object with respect to the form. This parameter is used only when the event occurs in an object. The character position in the upper left corner of a form is 1; the x position increases by 1 for each character, counting from left to right. Include the following variable declaration:

```
VAR object_x_position : integer;
```

object_y_position {output}

A data name to receive the y position of the object with respect to the form. This parameter is used only when the event occurs in an object. The character position in the upper left corner of a form is 1; the y position increases by 1 for each character, counting from top to bottom. Include the following variable declaration:

```
VAR object_y_position : integer;
```

last_event {output}

Indicates whether this is the last event. If this is the last event, the value is T; if this is not the last event, the value is F. Include the following variable declaration:

```
VAR last_event : string(1);
```

status {output}

The record that indicates the results of the procedure.

Conditions fde$bad_data_value

Remarks The FDP$XREAD_FORMS procedure deletes existing events. If the event is normal, Screen Formatting updates the variables in the added and combined forms containing the event. Later, you can request the transfer of these variables to program storage. If the event is abnormal, Screen Formatting does not update or validate variables.

Getting a Real Variable

Purpose FDP$XGET_REAL_VARIABLE gets a value the user entered on a form for a real variable and transfers it to the program.

Format **FDP$XGET_REAL_VARIABLE (form_identifier, name, occurrence, variable, variable_status, status)**

Parameters form_identifier {input}

The identifier established when the form was opened. Include the following variable declaration:

VAR form_identifier : integer;

name {input}

The name of the variable to get. This name was defined when the form was created.

occurrence {input}

The occurrence of the variable name. Use 1 for the first or only occurrence. Include the following variable declaration:

VAR occurrence : integer;

variable {output}

The value of the real variable that Screen Formatting generates automatically in the form definition record. If you do not want to use the automatically generated variable, include a variable of type real.

variable_status {output}

An ordinal that gives you the status of the variable. The following values are possible:

FDC$INDEFINITE

The user entered an indefinite number.

FDC$INVALID_BDP_DATA

The user entered data that does not correspond to the defined data type.

FDC$INVALID_REAL

The user entered data that is not within the range of real numbers defined for the variable.

FDC$LOSS_OF_SIGNIFICANCE

The user entered a number too large to be converted to the defined real or integer program type.

FDC$NO_ERROR

No error occurred on the variable.

FDC$OVERFLOW

The user entered an exponent that is too large.

FDC$UNDERFLOW

The user entered an exponent that is too small.

Include the following variable declaration:

```
VAR variable_status : integer;
```

status {output}

The record that indicates the results of the procedure. Include the following variable declaration:

```
VAR status : integer;
```

Conditions

fde$bad_data_value
fde$invalid_form_identifier
fde$invalid_variable_name
fde$no_space_available
fde$system_error
fde$unknown_occurrence
fde$unknown_variable_name

Remarks

- Before you get a real variable, you must open the form on which the user enters the value. If you get the variable after opening the form and before reading or replacing the variable on the form, the program returns the initial value specified by the form designer.

- If the form designer specifies data validation rules and error processing to display an error message or form, your program does not need to look at the variable status parameter.

  If the form designer specifies data validation rules and no error processing, the program must look at the variable status parameter.

  If the form designer specifies no data validation rules, the program must look at the variable status parameter.

Getting a String Variable

Purpose FDP$XGET_STRING_VARIABLE gets a value the user entered on a form for a string variable and transfers it to the program.

Format **FDP$XGET_STRING_VARIABLE (form_identifier, name, occurrence, variable, variable_status, status)**

Parameters

form_identifier {input}

The identifier established when the form was opened. Include the following variable declaration:

```
VAR form_identifier : integer;
```

name {input}

The name of the variable to get. The name was defined when the form was created.

occurrence {input}

The occurrence of the variable name. Use 1 for the first or only occurrence. Include the following variable declaration:

```
VAR occurrence : integer;
```

variable {output}

The variable that Screen Formatting generates automatically in the form definition record. The form definition record defines the variable. If you do not want to use the automatically generated variable, include a variable of the following type (* is the number of characters in the variable):

```
string ( * )
```

variable_status {output}

An ordinal that gives you the status of the variable. The following values are possible:

FDC$INVALID_STRING

The user entered data that does not match the strings defined for the variable.

FDC$NO_ERROR

No error occurred on the variable.

FDC$VARIABLE_TRUNCATED

The storage length of the VARIABLE parameter is not long enough.

Include the following variable declaration:

```
VAR variable_status : integer;
```

status {output}

The record that indicates the results of the procedure. Include the following variable declaration:

```
VAR status : integer;
```

Conditions

fde$bad_data_value
fde$invalid_form_identifier
fde$invalid_variable_name
fde$no_space_available
fde$system_error
fde$unknown_occurrence
fde$unknown_variable_name
fde$wrong_variable_name

Remarks

- Before you get a string variable, you must open the form on which the user enters the value. If you get the variable after opening the form and before reading or replacing the variable on the form, the program returns the initial value specified by the form designer.

- If the form designer specifies data validation rules and error processing to display an error message or form, your program does not need to look at the variable status parameter.

  If the form designer specifies data validation rules and no error processing, the program must look at the variable status parameter.

  If the form designer specifies no data validation rules, the program must look at the variable status parameter.

Opening a Form

Purpose FDP$XOPEN_FORM locates a form and prepares it for use by the program.

Format **FDP$XOPEN_FORM (form_name, form_identifier, status)**

Parameters form_name: {input}

The name of the form you want to open. Include the following variable declaration:

```
VAR form_name : string(31);
```

form_identifier {input-output}

The form identifier established for the form. Other Screen Formatting procedures use this identifier when referencing the form. Include the following variable declaration:

```
VAR form_identifier : integer;
```

status {output}

The record that indicates the results of the procedure. Include the following variable declaration:

```
VAR status : integer;
```

Conditions

fde$bad_data_value
fde$form_already_open
fde$form_not_ended
fde$form_requires_conversion
fde$invalid_form_identifier
fde$invalid_form_name
fde$no_space_available
fde$system_error
fde$terminal_not_identified
fde$unknown_form_name

Remarks

- Screen Formatting locates a form as follows:

  - If the form name is blank, Screen Formatting assumes that the form identifier specifies the required dynamically created form.

  - If the form name is not blank, Screen Formatting searches the list of ended dynamically created forms.

  - If the form name is not blank and is not in the list of ended dynamically created forms, Screen Formatting searches the command library list to find the form name on the object libraries. (You specify the order in which Screen Formatting searches the list using the NOS/VE command CREATE_COMMAND_LIST_ENTRY).

- Executing FDP$XOPEN_FORM does not display the form on the screen. (See Reading a Form or Showing a Form.)

- The form identifier that FDP$XOPEN_FORM returns identifies the instance of open for a form. Forms dynamically created have only one instance of open. Forms stored on object libraries can have more than one instance of open. For each instance of open, Screen Formatting maintains the working environment (current value of variables and their display attributes) of the form.

Popping a Form

Purpose FDP$XPOP_FORMS deletes forms scheduled (added or combined) since the last FDP$XPUSH_FORMS call.

Format **FDP$XPOP_FORMS (status)**

Parameters **status** {output}

The record that indicates the results of the procedure. Include the following variable declaration:

```
VAR status : integer;
```

Conditions fde$bad_data_value
fde$no_forms_to_pop

Remarks Events associated with the last list of pushed forms become active.

Positioning a Form

Purpose

FDP$XPOSITION_FORM schedules moving a form to a new location. Using this procedure, you can define a form at one location and display it at another location, or you can move a form from where it is currently displayed to a new location.

Format

FDP$XPOSITION_FORM (form_identifier, screen_x_position, screen_y_position, status)

Parameters

form_identifier {input}

The form identifier established when the form was opened. Include the following variable declaration:

```
VAR form_identifier : integer;
```

screen_x_position {input}

The x position on the screen. The character position in the upper left corner of the screen is 1, and the x position increases by 1 for each character counting from left to right. Include the following variable declaration:

```
VAR screen_x_position : integer;
```

screen_y_position {input}

The y position on the screen. The character position in the upper left corner of the screen is 1, and the y position increases by 1 for each character counting from top to bottom. Include the following variable declaration:

```
VAR screen_y_position : integer;
```

status {output}

The record that indicates the results of the procedure. Include the following variable declaration:

```
VAR status : integer;
```

Conditions

fde$bad_data_value
fde$form_pushed
fde$form_too_large_for_screen
fde$invalid_form_identifier
fde$no_space_available
fde$system_error

Remarks

- When the program calls either the FDP$XREAD_FORMS or FDP$XSHOW_FORMS procedure, Screen Formatting displays the form on the screen at the position specified in the call to FDP$XPOSITION_FORM.

- If you call this procedure while the form is displayed, the form is deleted from its current location and added at the new location. The added form is displayed on top of any other form occupying the same area on the screen.

- If you call this procedure before the form is displayed, the form is displayed at the specified location.

- Before you position a form, you must open it.

- You cannot position a pushed form.

Pushing a Form

Purpose

FDP$XPUSH_FORMS causes Screen Formatting to record added and combined forms so you can return to them later.

Format

FDP$XPUSH_FORMS (status)

Parameters

status {output}

The record that indicates the results of the procedure. Include the following variable declaration:

```
VAR status : integer;
```

Conditions

fde$bad_data_value
fde$no_forms_to_push

Remarks

- Events associated with these forms are not passed to the program.
- A program cannot change or close a pushed form.
- Pushed forms are displayed on the screen. If you want newly added forms to appear on a blank screen, first add a blank form that covers the screen.

  Updates to the screen continue to show the pushed forms.
- This subroutine deactivates the events associated with forms scheduled for display (added or combined) since th last push call.

Reading a Form

Purpose

FDP$XREAD_FORMS updates the terminal screen and accepts input from the application user.

Format

FDP$XREAD_FORMS (status)

Parameters

status {output}

The record that indicates the results of the procedure. Include the following variable declaration:

```
VAR status : integer;
```

Conditions

fde$bad_data_value
fde$no_events_active
fde$no_forms_to_read
fde$system_error
fde$terminal_disconnected

Remarks

- A call to FDP$XREAD_FORMS:
  - Displays all the forms you scheduled for display and have not deleted. If you added or combined forms since the last FDP$XREAD_FORMS or FDP$XSHOW_FORMS call, it displays them for the first time.
  - Removes from the screen the forms you deleted since the last FDP$XREAD_FORMS or FDP$XSHOW_FORMS call.
  - Updates on the screen the variables replaced since the last FDP$XREAD_FORMS or FDP$XSHOW_FORMS call.
  - Updates on the screen the objects for which display attributes were set or reset since the last FDP$XREAD_FORMS or FDP$XSHOW_FORMS call.

- Events not retrieved with the FDP$XGET_NEXT_EVENT procedure are deleted before any input is accepted from the user.

- The FDP$XREAD_FORMS procedure does not execute unless the forms scheduled for display contain at least one active event.

- After issuing this request, your program does not regain control until the user issues a normal event and Screen Formatting validates all the data, or the user issues an abnormal event.

Replacing an Integer Variable

Purpose FDP$XREPLACE_INTEGER_VARIABLE transfers a program variable to Screen Formatting.

Format **FDP$XREPLACE_INTEGER_VARIABLE (form_identifier, name, occurrence, variable, variable_status, status)**

Parameters form_identifier {input}

The identifier established when the form was opened. Include the following variable declaration:

```
VAR form_identifier : integer;
```

name {input}

The name of the variable to replace. This name was defined when the form was created.

occurrence {input}

The occurrence of the variable name. Use 1 for the first or only occurrence. Include the following variable declaration:

```
VAR occurrence : integer;
```

variable {input}

The integer variable that Screen Formatting generates automatically in the form definition record. If you do not want to use the automatically generated variable, include a variable of type integer.

variable_status {output}

An ordinal that gives you the status of the variable. The following values are possible:

FDC$LOSS_OF_SIGNIFICANCE

The program supplied a value that is too large for the form field.

FDC$NO_ERROR

No error occurred on the variable.

FDC$OUTPUT_FORMAT_BAD

The output format defined for the variable cannot output the variable.

Include the following variable declaration:

VAR variable_status : integer;

status {output}

The record that indicates the results of the procedure. Include the following variable declaration:

VAR status : integer;

Conditions

fde$bad_data_value
fde$form_pushed
fde$invalid_form_identifier
fde$invalid_variable_name
fde$no_space_available
fde$system_error
fde$unknown_occurrence
fde$unknown_variable_name
fde$wrong_variable_type

Remarks

- When you call either the FDP$XREAD_FORMS or FDP$XSHOW_FORMS procedure, Screen Formatting replaces the integer variable on the terminal screen.
- Before you replace an integer variable, you must open the form on which it is replaced.
- You cannot replace an integer variable for a pushed form.
- If the integer variable is not valid, it is not replaced.

Replacing a Real Variable

Purpose

FDP$XREPLACE_REAL_VARIABLE transfers a real program variable to Screen Formatting.

Format

FDP$XREPLACE_REAL_VARIABLE (form_identifier, name, occurrence, variable, variable_status, status)

Parameters

form_identifier {input}

The identifier established when the form was opened. Include the following variable declaration:

```
VAR form_identifier : integer;
```

name {input}

The name of the variable to replace. This name was defined when the form was created.

occurrence {input}

The occurrence of the variable name. Use 1 for the first or only occurrence. Include the following variable declaration:

```
VAR occurrence : integer;
```

variable {input}

The value of the real variable that Screen Formatting generates automatically in the form definition record. If you do not want to use the automatically generated variable, include a variable of type real.

variable_status {output}

An ordinal that gives you the status of the variable. The following values are possible:

FDC$LOSS_OF_SIGNIFICANCE

The value the program supplied is too large for the form variable.

FDC$NO_ERROR

No error occurred on the variable.

FDC$OUTPUT_FORMAT_BAD

The output format defined for the variable cannot output the variable.

Include the following variable declaration:

VAR variable_status : integer;

status {output}

The record that indicates the results of the procedure. Include the following variable declaration:

VAR status : integer;

Conditions

fde$bad_data_value
fde$form_pushed
fde$invalid_form_identifier
fde$no_space_available
fde$system_error
fde$unknown_occurrence
fde$unknown_variable_name
fde$variable_name
fde$wrong_variable_type

Remarks

- When you call either the FDP$XREAD_FORMS or FDP$XSHOW_FORMS procedure, Screen Formatting replaces the real variable on the terminal screen.
- Before you replace a real variable, you must open the form on which it is replaced.
- You cannot replace a real variable for a pushed form.
- If the real variable is not valid, it is not replaced.

Replacing a String Variable

Purpose

FDP$XREPLACE_STRING_VARIABLE transfers a program string variable to Screen Formatting.

Format

FDP$XREPLACE_STRING_VARIABLE (form_identifier, name, occurrence, variable, variable_status, status)

Parameters

form_identifier {input}

The identifier established when the form was opened. Include the following variable declaration:

```
VAR form_identifier : integer;
```

name {input}

The name of the variable to replace. This name was defined when the form was created.

occurrence {input}

The occurrence of the variable name. Use 1 for the first or only occurrence. Include the following variable declaration:

```
VAR occurrence : integer;
```

variable {input}

The variable that Screen Formatting generates automatically in the form definition record. The form definition record defines the variable. If you do not want to use the automatically generated variable, use a variable of the following type (* is the number of characters in the variable):

```
string ( * )
```

variable_status {output}

An ordinal that gives you the status of the variable. The following value is possible:

FDC$NO_ERROR

No error occurred on the variable.

Include the following variable declaration:

```
VAR variable_status : integer;
```

status {output}

The record that indicates the results of the procedure. Include the following variable declaration:

```
VAR status : integer;
```

Conditions

fde$bad_data_value
fde$form_pushed
fde$invalid_form_identifier
fde$invalid_variable_name
fde$no_space_available
fde$system_error
fde$unknown_occurrence
fde$unknown_variable_name
fde$wrong_variable_type

Remarks

- When the program calls either the FDP$XREAD_FORMS or FDP$XSHOW_FORMS procedure, Screen Formatting replaces the string variable on the terminal screen.
- Before you replace a string variable, you must open the form on which it is replaced.
- You cannot replace a string variable for a pushed form.
- If the string variable is not valid, it is not replaced.
- If the form specifies that the data must be in upper case, Screen Formatting converts it to upper case before storing the data in the form.

Resetting a Form

Purpose

FDP$XRESET_FORM resets the form to the state specified by the form definition.

Format

FDP$XRESET_FORM (form_identifier, status)

Parameters

form_identifier {input}

The identifier established when the form was opened. Include the following variable declaration:

```
VAR form_identifier : integer;
```

status {output}

The record that indicates the results of the procedure. Include the following variable declaration:

```
VAR status : integer;
```

Conditions

fde$bad_data_value
fde$form_pushed
fde$invalid_form_identifier
fde$no_space_available
fde$system_error

Remarks

- When the program calls either the FDP$XREAD_FORMS or FDP$XSHOW_FORMS procedure, Screen Formatting displays the form on the terminal screen with the reset specifications.
- All variables belonging to the form have their initial values and display attributes. The form is in its defined position.
- Before you reset a form, you must open it.
- You cannot reset a pushed form.

Resetting an Object Attribute

Purpose

FDP$XRESET_OBJECT_ATTRIBUTE resets the display attributes for an object to those specified in the form definition.

Format

FDP$XRESET_OBJECT_ATTRIBUTE (form_identifier, object_name, occurrence, status)

Parameters

form_identifier {input}

The identifier established when the form was opened. Include the following variable declaration:

```
VAR form_identifier : integer;
```

object_name {input}

The name of the object whose attributes are being reset. This name was defined when the form was created. Include the following variable declaration:

```
VAR object_name : string(31);
```

occurrence {input}

The occurrence of the object. For the first or only occurrence, use 1. Include the following variable declaration:

```
VAR occurrence : integer;
```

status {output}

The record that indicates the results of the procedure. Include the following variable declaration:

```
VAR status : integer;
```

Conditions

fde$bad_data_value
fde$form_not_scheduled
fde$form_pushed
fde$invalid_form_identifier
fde$invalid_object_name
fde$invalid_occurrence
fde$no_space_available
fde$unknown_object_name

Remarks

- You can reset the attributes of objects that are variable text, constant text, lines, or boxes.
- Before you reset the attribute of an object, you must open and either add or combine the form the object is on.
- When the program calls either the FDP$XREAD_FORMS or FDP$XSHOW_FORMS procedure, Screen Formatting displays the object using the reset attributes.

Setting the Cursor Position

Purpose FDP$XSET_CURSOR_POSITION sets the cursor to a selected position for later display.

Format **FDP$XSET_CURSOR_POSITION (form_identifier, object_name, occurrence, character_position, status)**

Parameters form_identifier {input}

The identifier established when the form was opened. Include the following variable declaration:

VAR form_identifier : integer;

object_name {input}

The name of the object on which you want to set the cursor. This name was defined when the form was created. Include the following variable declaration:

VAR object_name : string(31);

occurrence {input}

The integer specifying the occurrence of the object name. Use 1 for the first occurrence. Include the following variable declaration:

VAR occurrence : integer;

character_position {input}

The character position to which you want to set the cursor. Use 1 for the first character position. Include the following variable declaration:

VAR character_position : integer;

status {output}

The record that indicates the results of the procedure. Include the following variable declaration:

VAR status : integer;

Conditions

fde$bad_data_value
fde$form_not_scheduled
fde$form_pushed
fde$invalid_character_position
fde$invalid_form_identifier
fde$invalid_object_name
fde$no_object_available_defined
fde$no_space_available
fde$system_error
fde$unknown_object_name
fde$unknown_occurrence

Remarks

- One use of this procedure is to alter the default sequence of the application user's entry of variables. In the default sequence, Screen Formatting places the cursor on the first input variable of the highest priority form. The first character of the highest priority form is the form last added, combined, or positioned.

  At terminals with protected fields, the user then tabs from one variable text object to the next. The cursor starts at the top line of the form. It moves from left to right on each line. When no variable text object appears on a line, the cursor moves down to the next line. At terminals without protected fields, the user must move the cursor using the arrow keys or the tab and return keys.

- When you call either the FDP$XREAD_FORMS or FDP$XSHOW_FORMS procedure, Screen Formatting updates the terminal screen with the cursor at the specified position.

- If the position you specify is not visible on the screen, Screen Formatting shifts the data to make the cursor visible.

- The cursor position is in effect only for the next screen update from reading or showing forms.

- Before you set the cursor position on a form, you must open the form and either add or combine it.

- You cannot set the cursor position in a pushed form.

Setting Line Mode

Purpose

FDP$XSET_LINE_MODE begins line-by-line interaction with an application user.

Format

FDP$XSET_LINE_MODE (status)

Parameters

status {output}

The record that indicates the results of the procedure. Include the following variable declaration:

```
VAR status : integer;
```

Conditions

fde$bad_data_value

Remarks

- Use this call for extended dialogues in line mode. For short dialogues, Screen Formatting automatically switches to the proper mode (line or screen) but resources used for screen mode interaction remain.
- This call releases all screen mode resources:
  - Open forms are closed.
  - The mode is set to line.

Setting an Object Attribute

Purpose

FDP$XSET_OBJECT_ATTRIBUTE changes a display attribute for an object.

Format

FDP$XSET_OBJECT_ATTRIBUTE (form_identifier, object_name, occurrence, attribute, status)

Parameters

form_identifier {input}

The identifier established when the form was opened. Include the following variable declaration:

```
VAR form_identifier : integer;
```

object_name {input}

The name of the object whose display attribute is being reset. Include the following variable declaration:

```
VAR object_name : string(31);
```

occurrence {input}

The occurrence of the object. For the first or only occurrence, use 1. Include the following variable declaration:

```
VAR occurrence : integer;
```

attribute {input}

The name given to the display attribute when it was defined on the form. The attribute used here is defined for the form and not for a specific object. When using Screen Design Facility, screen attributes are defined through the ATTRIB function. When using a CYBIL program, the ADD_DISPLAY_DEFINITION attribute record defines form attributes.

Include the following variable declaration:

```
VAR attribute : string(31);
```

status {output}

The record that indicates the results of the procedure. Include the following variable declaration:

```
VAR status : integer;
```

Conditions

fde$bad_data_value
fde$form_not_scheduled
fde$form_pushed
fde$invalid_attribute_position
fde$invalid_form_identifier
fde$invalid_object_name
fde$invalid_occurrence
fde$no_space_available
fde$unknown_display_name
fde$unknown_object_name
fde$unknown_occurrence

Remarks

- You can set the attributes of objects that are variable text, constant text, lines, or boxes.
- Changed attributes replace existing attributes.
- When you call either the FDP$XREAD_FORMS or FDP$XSHOW_FORMS procedure, Screen Formatting displays the object using the set attributes.
- If the object you specify is not visible on the screen, Screen Formatting shifts the data to make the object visible.
- Before you set the attribute of an object, you must open the form the object is on and either add or combine it.
- You cannot set attributes of objects on a pushed form.

Showing Forms

Purpose FDP$XSHOW_FORMS updates the terminal screen.

Format **FDP$XSHOW_FORMS (status)**

Parameters status {output}

A record that indicates the results of the procedure. Include the following variable declaration:

```
VAR status : integer;
```

Conditions

fde$bad_data_value
fde$form_too_large_for_screen
fde$form_to_show
fde$no_space_available
fde$system_error
fde$terminal_disconnected

Remarks

- When none of the forms scheduled for display has an event or input variable defined, use this procedure instead of FDP$XREAD_FORMS.

- When you do not want any input from the terminal user, use this procedure.

- A call to FDP$XSHOW_FORMS:

  - Displays all the forms you have scheduled for display and have not deleted. If you added or combined forms since the last FDP$XREAD_FORMS or FDP$XSHOW_FORMS call, it displays them for the first time.

  - Removes from the screen the forms you deleted since the last FDP$XREAD_FORMS or FDP$XSHOW_FORMS call.

  - Displays variables replaced since the last FDP$XREAD_FORMS or FDP$XSHOW_FORMS call.

  - Displays objects with attributes set or reset since the last FDP$XREAD_FORMS or FDP$XSHOW_FORMS call.

Using CYBIL to Manage Forms 6

Writing a Program to Use Forms 6-2
Copying Procedure Definitions 6-2
Copying Data Definitions 6-3
Calling Screen Formatting 6-4
Displaying and Removing Forms and Variable Data 6-4
Processing Events and Data 6-6
Processing Normal Events 6-6
Processing Abnormal Events 6-7
Running a Prototype of the Application 6-7
Example Program for Managing Forms with CYBIL 6-9
Forms Managed in the Program 6-9
Design Specification 6-12
Form Definition Decks 6-14
Example CYBIL Program 6-15

Expanding and Compiling a Program 6-25

Helping the User Start the Application 6-27
Creating a User Procedure 6-27
Creating a User Prolog 6-28
Selecting a Natural Language 6-29
Starting the Application 6-29

CYBIL Procedure Calls for Interacting with Forms 6-30
Adding a Form 6-31
Changing Table Size 6-32
Closing a Form 6-34
Combining Forms 6-35
Deleting a Form 6-37
Getting an Integer Variable 6-38
Getting the Next Event 6-40
Getting a Real Variable 6-43
Getting a Record 6-45
Getting a String Variable 6-47
Opening a Form 6-49
Popping a Form 6-51
Positioning a Form 6-52
Pushing a Form 6-54
Reading a Form 6-55
Replacing an Integer Variable 6-57
Replacing a Real Variable 6-59
Replacing a Record 6-61
Replacing a String Variable 6-63
Resetting a Form 6-65

Resetting an Object Attribute 6-66
Setting the Cursor Position 6-67
Setting Line Mode 6-69
Setting an Object Attribute 6-70
Showing Forms 6-72

Using CYBIL to Manage Forms 6

Chapter 1 presented an example of creating and managing forms. It demonstrated that both the designer and the programmer have specific tasks to accomplish. When creating forms, and then managing the forms using a CYBIL program, the following tasks need to be accomplished:

1. The form designer and programmer plan the forms and program.

2. The form designer creates the forms specifying CYBIL as the form processor (or programming language) and prepares a design specification.

3. The form designer puts the forms in an object library and makes the form record definition available. Each record definition contains the data definitions of all variables defined on a particular form and is written in CYBIL.

4. The programmer codes the program, including calls to Screen Formatting CYBIL subroutines based on the design specification. These calls manage the forms created by the designer.

5. The programmer expands and compiles the program.

6. The programmer writes a user procedure to start the application and helps the user set up the correct terminal environment for using the forms.

When the last task is complete, the program and forms are ready for the application user.

This chapter describes the tasks performed by the programmer and shows them being executed in a CYBIL program. At the end of the chapter you will find format and parameter descriptions for each call to CYBIL subroutines used by Screen Formatting.

The designer's tasks and, also, the formats of the CYBIL procedure calls that create forms are described in chapter 7. (For information about designing forms using the Screen Design Facility, see the NOS/VE Screen Design Facility manual instead.)

Writing a Program to Use Forms

When writing a program to use forms, you must:

- Copy the procedure definitions for the CYBIL procedures used by Screen Formatting.
- Copy the data definitions generated by Screen Formatting when the designer creates the form. The data definitions hold values transferred to and from the form for the variable text objects.
- Call Screen Formatting procedures to manage the forms and the variable text objects on the forms.

Following the descriptions of these tasks is a CYBIL program in which these tasks are executed.

Copying Procedure Definitions

The procedure definitions define the procedures and their parameters. For every procedure used in the program, you must copy the procedure definition using the SCU *COPYC directive.

Copying Data Definitions

The data definitions for each form reside on a *form definition record* created by the form designer. In your program, you transfer data to and from variable text objects through this record.

When the designer creates a form, Screen Formatting generates a common deck that defines the form definition record. For example, Screen Formatting[1] generated the following source file for a form named CYBIL-SELECT-FORM. (The form definition record name is the same as the form name.)

```
*DECK CYBIL_SELECT_FORM expand = false
  TYPE
    cybil_select_form = record
      align_field: ALIGNED [0 MOD 8] string (0),
      message: string (40),
      object: string (1),
    recend;
```

The designer saves this file as a deck on a NOS/VE source library using the SOURCE_CODE_UTILITY (SCU).[2] The DECK directive in the file creates the correct name for the deck when it is processed.

In the beginning of your program, you must copy the form definition deck for each form created by the designer:

- Get the name of the deck from the design specification (the designer assigns the name while creating the form).
- Copy the deck by specifying its name on the SCU *COPY directive.

1. For this example, Screen Formatting was accessed through the Screen Design Facility.

2. Because each form has its own definition and the STATUS parameters use common decks, we recommend that you manage the source text using SCU. (For information on SCU, see the NOS/VE Source Code Management manual.)

Calling Screen Formatting

When you write a program that uses forms, you perform two basic tasks with Screen Formatting procedures:

- Displaying and removing forms and variable data on the application user's screen.
- Processing events executed by the user.

Displaying and Removing Forms and Variable Data

To control the display of forms and variable data on the user's screen, you perform the following steps in the sequence given:

1. Open the form.

   When you open a form, Screen Formatting locates it and allocates resources for processing the Screen Formatting calls that use the form.

   You need open a form only once, no matter how many times you use or update it. For this reason, begin a procedure by opening all the forms you will use. When a form requires a large amount of storage for variables, however, you may want to open that one only when the application user needs it.

   (For the format of the call that opens forms, see *Opening a Form* later in this chapter).

2. Add the form.

   When you add a form, Screen Formatting schedules it for display on the application user's screen.

   To display more than one form at a time, add all the forms before you display them. Screen Formatting maintains a list of all forms you add. The last form you add to the list becomes the top form on the screen. Because forms are opaque, the top form covers other forms appearing in the same area.

   When the terminal user completes data entry, the cursor position indicates what form Screen Formatting should process. Variables on this form (and any forms combined with this one) are validated and updated. Variables on other forms are not updated or validated.

   (For the formats of the calls that schedule forms for display, see *Adding a Form* and *Combining Forms* later in this chapter.)

3. Read the form.

   When you read a form, Screen Formatting displays all the forms you've added.

   When a form has an event or input variable defined, reading forms also accepts data from the application user and displays values returned by the program.

   (For the format of the call that reads forms, see *Reading Forms* later in this chapter. When none of the forms scheduled for display has an event or input variable defined, you can use a similar call described in *Showing Forms* later in this chapter.)

4. Delete the form.

   When you delete a form, Screen Formatting deletes it from the list of forms scheduled for display. The next time you read forms, the deleted form is removed from the screen. However, the form remains available for later use in the program (you must reschedule it for display).

   (For the format of the call that deletes a form, see *Deleting a Form* later in this chapter.)

5. Close the form.

   When you close a form, Screen Formatting releases the resources the form uses. The form is no longer available to the user or your program.

   (For the format of the call that closes a form, see *Closing a Form* later in this chapter.)

Processing Events and Data

When creating a form, the designer defines two types of events a user can execute to return control to the program: normal and abnormal.

- For normal events, the program performs requested actions such as getting variables, doing computations, and updating the form.
- For abnormal events, the program takes its own action. You generally then delete the form and go on, or stop the program.

Processing Normal Events

To process a normal event:

1. Get the name of the event and the position of the cursor from Screen Formatting.

   Screen Formatting validates the data the user enters (the form designer defined the validation rules) and transfers values of screen variables to its storage. The form designer may also have created error forms to be displayed when the user enters an incorrect value or presses a key not defined as an event.

   (For the format of the call that gets the event name and cursor position, see *Getting the Next Event* at the end of this chapter.)

2. Get the data from Screen Formatting storage and transfer it to program storage.

   (For formats of the calls that get data, see the following sections later in this chapter: *Getting a Record, Getting an Integer Variable, Getting a Real Variable,* and *Getting a String Variable.*)

3. Replace the data in Screen Formatting storage with the data in program storage.

   (For formats of the calls that replace variables, see the following sections later in this chapter: *Replacing a Record, Replacing an Integer Variable, Replacing a Real Variable,* and *Replacing a String Variable.*)

You can also reset the variables on a form to their original state. (For formats of the calls that reset variables to their original state, see *Resetting a Form* and *Resetting an Object Attribute* later in this chapter.)

Processing Abnormal Events

To process an abnormal event:

1. Get the name of the event and the position of the cursor from Screen Formatting.

   Unlike a normal event, Screen Formatting neither validates user entries nor transfers values of screen variables to Screen Formatting storage.

   (For the format of the call that gets the event name and cursor position, see *Getting the Next Event* later in this chapter.)

2. Write your own procedure to perform the task the design specification assigns to the event. Typical actions for an abnormal event include:

   - Resetting a form and redisplaying it.
   - Moving the user to a new form for additional processing.
   - Returning the user to a previous form.
   - Stopping the program.

   The user's screen is updated when you either read the forms again or end the program.

Running a Prototype of the Application

Once the forms have been created for your application, you can interactively run a prototype using the MANAGE_FORMS utility. This allows you to test the order in which the forms appear, and to interact with the forms as the application user will do.

An example of an application prototype is given in chapter 2 under the section named The Application Prototype. This prototype uses forms that were created specifically for use in an SCL procedure. To learn about using a prototype, you can run the prototype as described in the section.

Once you are familiar with the utility, you can also run a prototype using forms created for a CYBIL program. Because the naming conventions for CYBIL do not conflict with SCL naming conventions, the variables defined for the form can be used in the prototype without any conversion taking place.

To use the prototype with the CYBIL forms that are on the library specified in the prototype example, rather than opening the SCL forms listed, open the forms named:

CYBIL_SELECT_FORM
CYBIL_RECTANGLE_FORM
CYBIL_CIRCLE_FORM

One variable of type RECORD is created for each form as described for the SCL forms shown in the prototype example. The variable has the same name as the form. You can display the data structure of each variable using the DISPLAY_VALUE command. For example, to display the data structure for the CYBIL_RECTANGLE_FORM variable, enter the following command:

```
mf/display_value value=cybil_rectangle_form ..
mf../display_options=data_structure
display option: DATA_STRUCTURE

"RECORD"
  SIDE_TABLE: "ARRAY"
              1. "RECORD"
                  SIDE: "INTEGER"  0
                 "RECORD END"

              2. "RECORD"
                  SIDE: "INTEGER"  0
                 "RECORD END"
             "ARRAY END"

  AREA: "INTEGER"  0
  MESSAGE: "STRING"  '
"RECORD END"
```

For more information about the structure of variables that are records, see the NOS/VE System Usage manual.

Example Program for Managing Forms with CYBIL

The program in this example computes the area of circles and rectangles. The example includes:

- Pictures of the forms managed in the program.
- The design specification supplied by the form designer.
- The form definition decks.
- The example program.

Forms Managed in the Program

The example program manages three forms residing on an object library named EXAMPLE_OBJECT_LIBRARY that must be in the user's command list.

When a user starts the application, Select Form appears (figure 6-1).

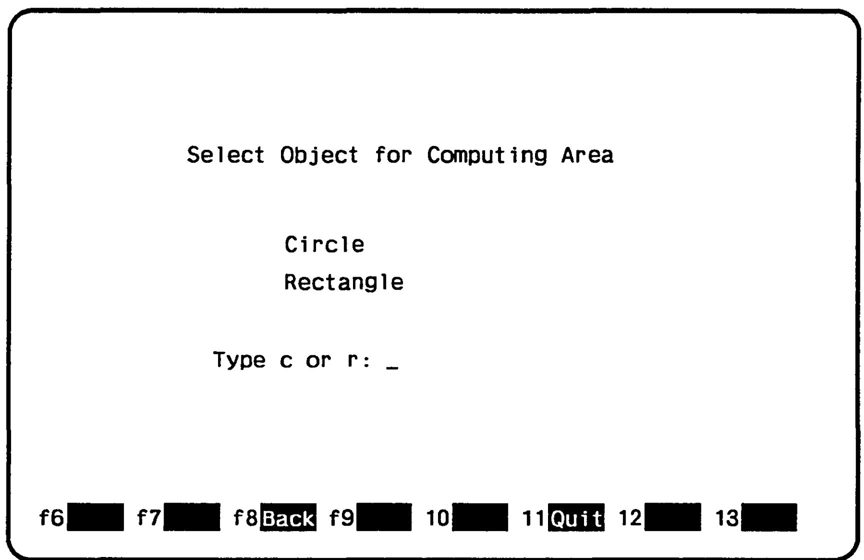


Figure 6-1. Select Form

On Select Form, a user enters either *c* to compute the area of a circle or *r* to compute the area of a rectangle.

When a user enters *r* on Select Form, Rectangle Form (figure 6-2) appears.

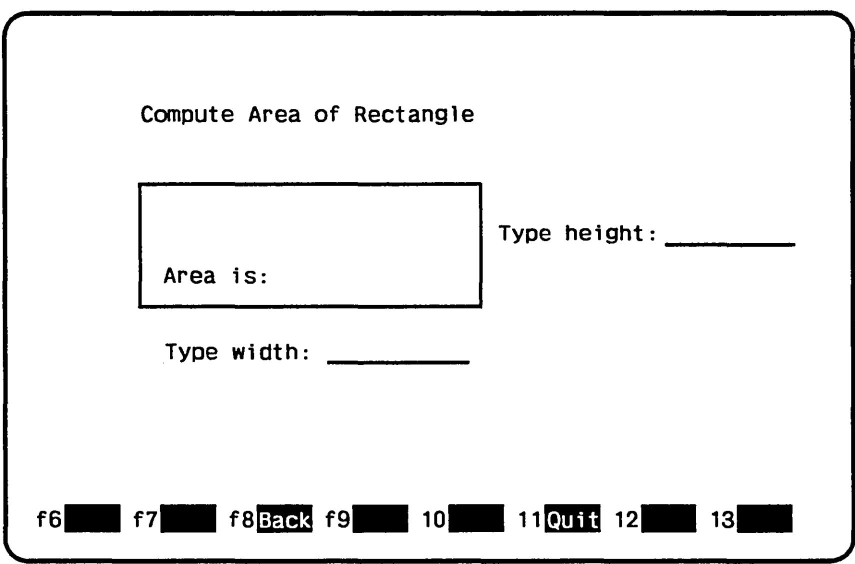


Figure 6-2. Rectangle Form

On Rectangle Form, the user enters the lengths of the sides of the rectangle as integers and presses the return key to have the program compute the area.

When a user enters *c* on Select Form, Circle Form (figure 6-3) appears.

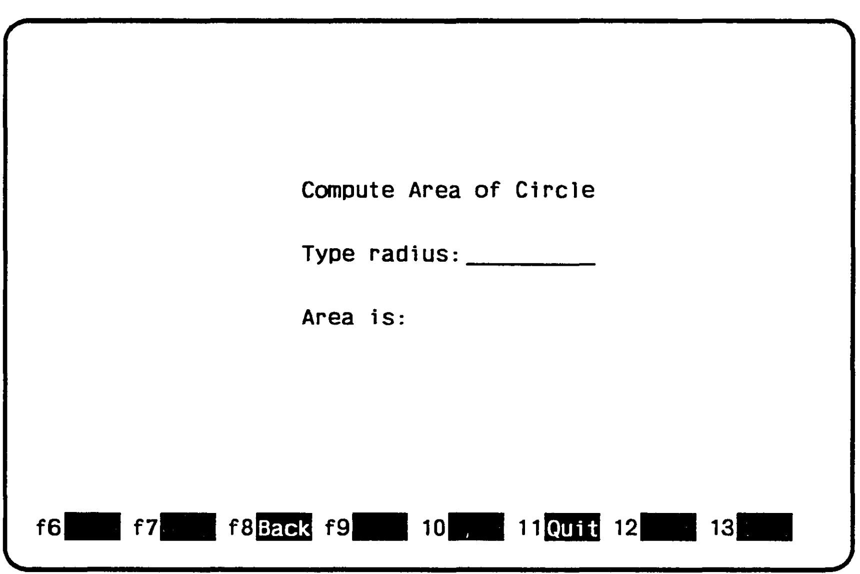


Figure 6-3. Circle Form

On Circle Form, the user enters the radius of the circle as a real value and presses the return key to have the program compute the area.

Design Specification

In writing the example program, the programmer uses the information the form designer listed in the following design specification:

- The names for the three forms used by the program are:

  CYBIL_SELECT_FORM
  CYBIL_RECTANGLE_FORM
  CYBIL_CIRCLE_FORM

- The user can call both the Rectangle Form and Circle Form from the Select Form.

- The following variable text objects are defined on the forms:

Variable Object	Description
Select Form:	
MESSAGE	Area for displaying error messages.
OBJECT	Area for user input of *r* or *c*.
Rectangle Form:	
SIDE_TABLE	Table that holds values for the rectangle's sides.
SIDE	Areas (two) for user input of values for the rectangle's sides.
AREA	Area for returning value of computed area.
MESSAGE	Area for displaying error messages.
Circle Form:	
RADIUS	Area for user input of value for the circle's radius.
AREA	Area for returning value of computed area.
MESSAGE	Area for displaying error messages.

- The following events are defined on the forms:

Event	Description
COMPUTE	A normal program event that processes data the user entered on the form. For Select Form, the COMPUTE event checks whether the user entered *r* or *c* and then displays the appropriate form. For the other forms, COMPUTE calculates the area and redisplays the form.
BACK	An abnormal program event that takes the user back to a previous environment. For Select Form, the BACK event stops the program. For the other forms, BACK returns the user to Select Form.
QUIT	An abnormal program event that stops the program.

Form Definition Decks

When the designer creates the three forms (by writing a program or using Screen Design Facility), a form definition record is created with each form. For the example program, the programmer copies the following form definition decks placed by the designer on an SCU library. The library in this example is named EXAMPLE_SOURCE_LIBRARY.

The CYBIL_SELECT_FORM deck:

```
TYPE
  cybil_select_form = record
    align_field: ALIGNED [0 MOD 8] string (0),
    message: string (40),
    object: string (1),
  recend;
```

The CYBIL_RECTANGLE_FORM deck:

```
TYPE
  cybil_rectangle_form = record
    align_field: ALIGNED [0 MOD 8] string (0),
    side_table: array [1 .. 2] of record
      side: ALIGNED [0 MOD 8] integer,
    recend,
    area: ALIGNED [0 MOD 8] integer,
    message: string (40),
  recend;
```

The CYBIL_CIRCLE_FORM deck:

```
TYPE
  cybil_circle_form = record
    align_field: ALIGNED [0 MOD 8] string (0),
    area: ALIGNED [0 MOD 8] real,
    radius: ALIGNED [0 MOD 8] real,
    message: string (40),
  recend;
```

Example CYBIL Program

This CYBIL program calls the forms and executes the events described in the previous sections. The program is in the SCU deck named CYBIL_COMPUTE_OBJECT_AREA. To run the example program, see the Screen Formatting examples in the Examples online manual.

```
?? RIGHT := 110 ??
MODULE compute_object_area;

{ Copy definitions for Screen Formatting procedures.

*copyc fdp$add_form
*copyc fdp$close_form
*copyc fdp$delete_form
*copyc fdp$get_real_variable
*copyc fdp$get_integer_variable
*copyc fdp$get_next_event
*copyc fdp$get_string_variable
*copyc fdp$open_form
*copyc fdp$read_forms
*copyc fdp$replace_string_variable
*copyc fdp$replace_integer_variable
*copyc fdp$replace_real_variable
*copyc fdp$reset_form
*copyc fdp$reset_object_attribute
*copyc fdp$set_cursor_position
*copyc fdp$set_object_attribute

*copyc pmp$abort
*copyc pmp$exit

  VAR
    circle_form_identifier: fdt$form_identifier,
     display_name: [READ] ost$name := 'ERROR',
    event_name: ost$name,
    event_normal: boolean,
    event_position: fdt$event_position,
    form_name: ost$name,
    last_event: boolean,
    rectangle_form_identifier: fdt$form_identifier,
    select_form_identifier: fdt$form_identifier,
    status: ost$status,
    variable_name: ost$name,
    variable_status: fdt$variable_status;
```

```
  PROCEDURE [INLINE] check_status;

    IF NOT status.normal THEN
      pmp$abort (status);
    IFEND;

  PROCEDURE check_status;

  PROCEDURE display_variable_status
    (    message: string ( * );
     VAR form_identifier: fdt$form_identifier);

    variable_name := 'MESSAGE';
    fdp$replace_string_variable (form_identifier, variable_name,
          1, message, variable_status, status);
    check_status;

  PROCEND display_variable_status;

  PROCEDURE compute_circle_area;

{ Copy variables for circle form.

*copyc cybil_circle_form

    VAR
      circle_data: cybil_circle_form;

{ Display circle form in original state.

    fdp$reset_form (circle_form_identifier, status);
    check_status;
    fdp$add_form (circle_form_identifier, status);
    check_status;

{ Update screen and get radius from terminal user entry.

  /get_input/
    REPEAT
      fdp$read_forms (status);
      check_status;
```

```
      fdp$get_next_event (event_name, event_normal,
            event_position, last_event, status);
      check_status;

{ On BACK or QUIT event, return to caller.

      IF event_name <> 'COMPUTE' THEN
        fdp$delete_form (circle_form_identifier, status);
        check_status;
        RETURN;
      IFEND;

{ Remove any previous error message.

      display_variable_status (' ', circle_form_identifier);

{ Transfer terminal user entry for radius to program.

      variable_name := 'RADIUS';
      fdp$get_real_variable (circle_form_identifier,
            variable_name, 1, circle_data.radius,
            variable_status, status);
      check_status;
      IF variable_status <> fdc$no_error THEN
        fdp$set_cursor_position (circle_form_identifier,
              variable_name, 1, 1, status);
        display_variable_status ('Type valid value for radius.',
              circle_form_identifier);
        CYCLE /get_input/;
      IFEND;

{ Compute area of circle and display it.

      circle_data.area := 3.14 * (circle_data.radius *
            circle_data.radius);
      variable_name := 'AREA';
      fdp$replace_real_variable (circle_form_identifier,
            variable_name, 1, circle_data.area, variable_status,
            status);
      check_status;
       IF variable_status <> fdc$no_error THEN
```

```
{ Area value could not be displayed using
{ output format defined for form.
{ Revise the form or the program to accommodate
{ the size of the number.

          variable_name := 'RADIUS';
          fdp$set_cursor_position (circle_form_identifier,
               variable_name, 1, 1, status);
          display_variable_status ('Format cannot display area.',
               circle_form_identifier);
          CYCLE /get_input/;
        IFEND;

      UNTIL FALSE;

    PROCEND compute_circle_area;

    PROCEDURE compute_rectangle_area;

{ Copy variables for rectangle form.

*copyc cybil_rectangle_form

      VAR
        rectangle_data: cybil_rectangle_form;

{ Display rectangle form in original state.

      fdp$reset_form (rectangle_form_identifier, status);
      check_status;

      fdp$add_form (rectangle_form_identifier, status);
      check_status;

{ Update screen and get terminal user entry
{ for rectangle height and width.

    /get_input/
      REPEAT
        fdp$read_forms (status);
        check_status;
```

```
      fdp$get_next_event (event_name, event_normal,
            event_position, last_event, status);
      check_status;

{ If abnormal event (BACK or QUIT) occurs, return to caller.

      IF event_name <> 'COMPUTE' THEN
        fdp$delete_form (rectangle_form_identifier, status);
        check_status;
        RETURN;
      IFEND;

{ Remove any previous error indications.

      display_variable_status (' ', rectangle_form_identifier);
      variable_name := 'SIDE';
      fdp$reset_object_attribute (rectangle_form_identifier,
            variable_name, 1, status);
      fdp$reset_object_attribute (rectangle_form_identifier,
            variable_name, 2, status);

{ Transfer height value from form to program.

      variable_name := 'SIDE';
      fdp$get_integer_variable (rectangle_form_identifier,
            variable_name, 1, rectangle_data.side_table [1].side,
            variable_status, status);
      check_status;

{ If data invalid, move cursor to height value
{ and display error message.

      IF variable_status <> fdc$no_error THEN
        fdp$set_cursor_position (rectangle_form_identifier,
              variable_name, 1, 1, status);
         fdp$set_object_attribute (rectangle_form_identifier,
              variable_name, 1, display_name, status);
        display_variable_status ('Type valid value for height.',
              rectangle_form_identifier);
        CYCLE /get_input/;
      IFEND;
```

```
{ Transfer width value from form to program.

      fdp$get_integer_variable (rectangle_form_identifier,
            variable_name, 2, rectangle_data.side_table [2].side,
            variable_status, status);
      check_status;

{ If data invalid, move cursor to width value
{ and display error message.

      IF variable_status <> fdc$no_error THEN
        fdp$set_cursor_position (rectangle_form_identifier,
              variable_name, 2, 1, status);
        fdp$set_object_attribute (rectangle_form_identifier,
              variable_name, 2, display_name, status);
        display_variable_status ('Type valid value for width.',
              rectangle_form_identifier);
        CYCLE /get_input/;
      IFEND;

{ Compute area of rectangle and display it.

      rectangle_data.area := rectangle_data.side_table [1].side *
            rectangle_data.side_table [2].side;

      variable_name := 'AREA';
      fdp$replace_integer_variable (rectangle_form_identifier,
            variable_name, 1, rectangle_data.area,
            variable_status, status);
      check_status;
      IF variable_status <> fdc$no_error THEN
```

```
{ Area value could not be displayed using
{ output format defined for form.
{ Revise the form or the program to accommodate
{ the size of the number.

        display_variable_status ('Format cannot display area.',
              rectangle_form_identifier);
        CYCLE /get_input/;
      IFEND;

     UNTIL FALSE;

  PROCEND compute_rectangle_area;

  PROCEDURE stop_program;

{ Close all forms.

    fdp$close_form (select_form_identifier, status);
    check_status;

    fdp$close_form (circle_form_identifier, status);
    check_status;

    fdp$close_form (rectangle_form_identifier, status);
    check_status;

    status.normal := TRUE;
    pmp$exit (status);
  PROCEND stop_program;

  PROGRAM compute_object_area;

*copyc cybil_select_form

    VAR
      select_data: cybil_select_form;
```

```
{ Open all forms used by the program
{ and assign form identifiers.

    form_name := 'CYBIL_SELECT_FORM';
    fdp$open_form (form_name, select_form_identifier, status);
    check_status;

    form_name := 'CYBIL_CIRCLE_FORM';
    fdp$open_form (form_name, circle_form_identifier, status);
    check_status;

    form_name := 'CYBIL_RECTANGLE_FORM';
    fdp$open_form (form_name, rectangle_form_identifier, status);
    check_status;

{ Add select form to list scheduled for display.

    fdp$add_form (select_form_identifier, status);
    check_status;

{ Update screen and accept user terminal entry
{ for object; display all added forms.

  /get_input/
    REPEAT
      fdp$read_forms (status);
      check_status;

{ Get screen event(s) that determine next actions.

      fdp$get_next_event (event_name, event_normal,
            event_position, last_event, status);
      check_status;

{ Stop program on QUIT or BACK event.

      IF event_name <> 'COMPUTE' THEN
        stop_program;
      IFEND;
```

```
{ Transfer object variable from form to program.

      variable_name := 'OBJECT';
      fdp$get_string_variable (select_form_identifier,
            variable_name, 1, select_data.object,
            variable_status, status);
      check_status;
      IF variable_status <> fdc$no_error THEN
        display_variable_status ('Type c or r.',
              select_form_identifier);
      IFEND;

      IF select_data.object = 'R' THEN

{ Remove select form and compute area of rectangle.

        fdp$delete_form (select_form_identifier, status);
        check_status;
        compute_rectangle_area;

      ELSEIF select_data.object = 'C' THEN

{ Remove select form and compute area of circle.

        fdp$delete_form (select_form_identifier, status);
        check_status;
        compute_circle_area
      ELSE

{ If terminal user entered invalid data, display
{ error message and ask for another entry.

        display_variable_status ('Type c or r.',
              select_form_identifier);
        CYCLE /get_input/;
      IFEND;
```

```
{ Process event from rectangle form or circle form.

      IF event_name = 'QUIT' THEN
        stop_program;
      IFEND;

{ A BACK event occurred on rectangle form or circle form;
{ display select form in original state.

      fdp$reset_form (select_form_identifier, status);
      check_status;

      fdp$add_form (select_form_identifier, status);
      check_status;

    UNTIL FALSE;

  PROCEND compute_object_area;
MODEND compute_object_area;
```

Expanding and Compiling a Program

Programs using Screen Formatting use common decks and form definition records that reside outside the main program. To manage the source text for this type of program, put the program in one or more SCU decks. This allows you to update individual parts of a program and to use forms in more than one program without duplicating code.[3]

To expand and compile a program maintained in SCU decks:

1. Expand the deck containing the main program.

2. Compile the expanded program.

3. Put the compiled program on an object library.

A procedure for compiling and expanding a program is shown in the following example. (The example is based on the example program and form definition records described earlier. The example shows how to place decks on library EXAMPLE_SOURCE_LIBRARY.)

The procedure calls SCU to expand the SCU directives contained in the program. For this expansion, you must include the $SYSTEM.CYBIL.OSF$PROGRAM_INTERFACE library as an alternate base. The program is then compiled and put on an object library.

```
PROCEDURE cybil_compile_deck, cybcd (
  deck,  d: name = $required
  status)

  source_code_utility
    use_library base=example_source_library result=$null
    expand_deck deck=deck ..
    compile=$local.compile ..
    alternate_base=$system.cybil.osf$program_interface
  quit

  cybil input=$local.compile ..
    list=$local.listing runtime_checks=all  ..
     binary=$local.lgo  ..
    debug_aids=all
```

3. For information on SCU, see the NOS/VE Source Code Management manual.

```
    create_object_library
      add_module library=example_object_library
      combine_module library=$local.lgo
      generate_library library=example_object_library.$next
    quit

  PROCEND cybil_compile_deck
```

To use the procedure, put it on library EXAMPLE_OBJECT_LIBRARY and then add the library to your command list (using the CREATE_COMMAND_LIST_ENTRY command). You can execute the procedure by entering:

```
/cybil_compile_deck deck=cybil_compute_object_area
```

The compiled program is now also on library EXAMPLE_OBJECT_LIBRARY.

For more information on writing and using procedures, see the NOS/VE System Usage manual.

Helping the User Start the Application

The complete application consists of your program and the forms created by the designer. To integrate the forms with your program, you must:

- Create a procedure that gives users access to the object library containing the forms and program.
- Ensure that the user's terminal environment is set up properly to use the forms (in most instances, by creating a user prolog).
- Ensure that users select the correct natural language.
- Ensure that users know how to start the application.

Creating a User Procedure

To give the user access to the object library containing the forms:

1. Write a NOS/VE procedure from which the user starts the application.
2. Place the procedure on the library that contains the compiled program.

For example, the following procedure executes the application that uses the starting procedure COMPUTE_OBJECT_AREA on library EXAMPLE_OBJECT_LIBRARY. The other libraries accessed by the program are $SYSTEM.FDF$LIBRARY and $SYSTEM.TDU.TERMINAL_DEFINITIONS. Users must have these libraries available in order for the program to call the Screen Formatting procedures.

```
PROCEDURE cybil_compute_area, cybca (
  status)

  execute_task ..
    library=(example_object_library,$system.fdf$library,..
    $system.tdu.terminal_definitions) ..
    starting_procedure=compute_object_area

PROCEND cybil_compute_area
```

Creating a User Prolog

To ensure that the users' terminal environment is set up properly to use the forms, make sure they set the following terminal characteristics before they execute the procedure:

Characteristic	Description
Terminal model	Identifies the terminal to NOS/VE.
Attention character	Provides a character users can enter to interrupt the application.
Hold messages	Tells the network to hold all network messages until the user stops the application. Otherwise, a computer operator message may overwrite a form while a user is entering data, confusing the user.

In most instances, users should set up their terminal for the entire terminal session in their user prologs. The example below does the following:

- Identifies a Digital Equipment Corporation VT220 terminal to the system.
- Chooses the exclamation point as a way to interrupt the program.
- Holds all messages from a NAMVE/CDCNET network.
- Sets up the way the terminal uses the exclamation point to interrupt the program.

The users add the following commands to their user prologs:

```
change_terminal_attributes terminal_model=dec_vt220 ..
  attention_character='!' ..
  status_action=hold
change_term_conn_defaults attention_character_action=1
change_connection_attributes terminal_file_name=input aca=1
change_connection_attributes terminal_file_name=output aca=1
change_connection_attributes terminal_file_name=command aca=1
```

For a further explanation of how to interrupt a screen application during an interactive session, and what commands to use for networks other than NAMVE/CDCNET, see the NOS/VE System Usage manual.

Selecting a Natural Language

To ensure that users receive messages in the correct natural language, have them add the CHANGE_NATURAL_LANGUAGE command to their prologs. Because the default language is US_ENGLISH and all messages returned by Screen Formatting are in this language, have users include this command only when you have changed messages to another language.

Changing messages to other languages is described in the NOS/VE Object Code Management manual. The CHANGE_NATURAL_LANGUAGE command is described in the NOS/VE System Usage manual.

Starting the Application

To start the application, the users enter:

```
/create_command_list_entry e=example_object_library
/cybil_compute_area
```

When finished with the application, the users remove the object library from their command lists:

```
/delete_command_list_entry e=example_object_library
```

CYBIL Procedure Calls for Interacting with Forms

The following sections describe the CYBIL procedure calls to Screen Formatting modules. For each procedure, there is a purpose description, input format, list of parameters and their types, condition identifiers, and pertinent remarks.

An application program calls Screen Formatting procedures to interact with an application user through the use of forms. Each of these procedures is defined as a deck on the SCU library $SYSTEM.CYBIL.OSF$PROGRAM_INTERFACE. This library must be in the alternate base when compiling the application program.

These procedures are external routines that reside on the library called $SYSTEM.FDF$LIBRARY. This library must be in the user's program library list in order to execute the program.

For detailed information on CYBIL procedure calls, see the CYBIL Language Definition manual.

Adding a Form

Purpose FDP$ADD_FORM schedules a form for display on the application user's screen.

Format **FDP$ADD_FORM (form_identifier, status)**

Parameters form_identifier: fdt$form_identifier;

The identifier established when the form was opened.

status: VAR of ost$status;

The record that indicates the results of the procedure.

Conditions

fde$bad_data_value
fde$form_already_added
fde$form_pushed
fde$form_too_large_for_screen
fde$invalid_form_identifier
fde$no_space_available
fde$system_error

Remarks

- When you call either the FDP$READ_FORMS or FDP$SHOW_FORMS procedure, Screen Formatting displays the added form on the terminal screen. The added form is placed on top of other forms occupying the same area on the screen.

- When displayed, each form that is added operates independently from other forms that have been added. When a user executes a normal event, Screen Formatting validates and updates only those variables on the form associated with the event. To have forms share events, see Combining Forms later in this section.

- Before you add a form, you must open it.

- You cannot add a pushed form.

Changing Table Size

Purpose FDP$CHANGE_TABLE_SIZE changes the size of the table during program execution.

Format **FDP$CHANGE_TABLE_SIZE (form_identifier, table_name, table_size, status)**

Parameters form_identifier: fdt$form_identifier;

The identifier established when the form was opened.

table_name: ost$name;

The name of the table to change in size.

table_size: fdt$table_size;

The size of the table. While this procedure is in effect, Screen Formatting limits the number of stored occurrences allowed for a table to the value you specify on this parameter. How many occurrences are displayed at one time depends on the number of visible occurrences defined in the form.

If you specify zero for the table size, no occurrences appear on the form.

status: VAR of ost$status;

The record that indicates the results of the procedure.

Conditions fde$bad_data_value
fde$form_pushed
fde$invalid_form_identifier
fde$invalid_table_name
fde$invalid_table_size
fde$no_space_available
fde$unknown_table_name

Remarks

- The table must be present in an open form.
- The size limitation remains in effect until the next time you call the FDP$CHANGE_TABLE_SIZE procedure.
- The maximum size for a table is identified by the form as the maximum number of stored occurrences. If you specify a table size larger than the maximum, you receive an error message (fde$invalid_table_size).

Examples

The following examples describe how changing the size of a table affects the application user. On the form, the table's specifications are a maximum of 20 stored occurrences, of which 6 occurrences can be visible at one time.

- If you specify a table size of 10, Screen Formatting displays 6 occurrences and allows the application user to page to the 10th occurrence.
- If you specify a table size of 4, Screen Formatting displays 4 occurrences and does not allow the application user to page.

Closing a Form

Purpose FDP$CLOSE_FORM releases resources used to process a form and deletes the form from the list scheduled for display.

Format **FDP$CLOSE_FORM (form_identifier, status)**

Parameters form_identifier: fdt$form_identifier;

The identifier established when the form was opened.

status: VAR of ost$status;

The record that indicates the results of the procedure.

Conditions
fde$bad_data_value
fde$invalid_form_identifier
fde$form_pushed
fde$no_space_available

Remarks

- When the program calls either the FDP$READ_FORMS or FDP$SHOW_FORMS procedure, Screen Formatting removes the closed form from the terminal screen as a result of calling this procedure.

- Before you can close a form, you must open it.

- You cannot close a pushed form.

Combining Forms

Purpose FDP$COMBINE_FORM combines a form with a previously added form and schedules the combined form for display on the terminal screen.

Format **FDP$COMBINE_FORM (added_form_identifier, combine_form_identifier, status)**

Parameters **added_form_identifier**: fdt$form_identifier;

The identifier for this instance of the previously added form.

combine_form_identifier: fdt$form_identifier;

The identifier for the form you are combining with the previously added form.

status: VAR of ost$status;

The record that indicates the results of the procedure.

Conditions

fde$bad_data_value
fde$form_already_added
fde$form_already_combined
fde$form_pushed
fde$form_too_large_for_screen
fde$invalid_form_identifier
fde$no_space_available
fde$system_error

Remarks

- You cannot combine a pushed form.
- The combined form inherits the event definitions of the previously added form.
- Before you combine a form with a previously added form, you must open both forms.
- When the program calls either the FDP$READ_FORMS or FDP$SHOW_FORMS procedure, Screen Formatting displays the combined form. The combined form is placed on top of other forms occupying the same area on the screen.
- When the application user executes an event to return normally to the program, Screen Formatting updates all program variables associated with both the added and combined forms.
- To combine several forms with a previously added form, call this procedure more than once.

Deleting a Form

Purpose FDP$DELETE_FORM deletes the form from the list of forms scheduled for display.

Format **FDP$DELETE_FORM (form_identifier, status)**

Parameters form_identifier: fdt$form_identifier;

The identifier established when the form was opened.

status: VAR of ost$status;

The record that indicates the results of the procedure.

Conditions

fde$bad_data_value
fde$form_not_scheduled
fde$form_pushed
fde$invalid_form_identifier
fde$no_space_available

Remarks

- When the program calls either the FDP$READ_FORMS or FDP$SHOW_FORMS procedure, Screen Formatting removes the deleted form from the terminal screen and replots any forms uncovered by the deleted form.
- When you add a form (FDP$ADD_FORM) again that you previously deleted, the data in the form is retained.
- Before you delete a form, you must open it.
- You cannot delete a pushed form.
- If the form was added and has any combined forms associated with it, the combined forms are also deleted.
- When you delete a combined form, only that form is deleted. Areas covered by the combined form are replotted after the combined form is deleted.

Getting an Integer Variable

Purpose FDP$GET_INTEGER_VARIABLE gets the value the user entered on the form for an integer variable and transfers it to the program.

Format **FDP$GET_INTEGER_VARIABLE (form_identifier, name, occurrence, variable, variable_status, status)**

Parameters form_identifier: fdt$form_identifier;

The identifier established when the form was opened.

name: ost$name;

The name of the integer variable to get and transfer to the program. This name was defined when the form was created.

occurrence: fdt$occurrence;

The occurrence of the variable name. The values allowed are 1 .. 1000. Use 1 for the first or only occurrence.

variable: VAR of integer;

The integer variable that Screen Formatting generates automatically in the form definition record. If you do not want to use the automatically generated variable, use a variable of type integer.

variable_status: VAR of fdt$variable_status;

The condition name that describes the status of the integer variable.

FDC$INVALID_BDP_DATA

The user entered data that does not correspond to the defined program data type.

FDC$INVALID_INTEGER

The user entered data that is not in the range defined for the variable.

FDC$LOSS_OF_SIGNIFICANCE

The user entered an integer that is too large.

FDC$NO_ERROR

No error occurred on the variable.

status: VAR of ost$status;

The record that indicates the results of the procedure.

Conditions

fde$bad_data_error
fde$invalid_form_identifier
fde$invalid_variable_name
fde$no_space_available
fde$system_error
fde$unknown_occurrence
fde$unknown_variable_name
fde$wrong_variable_type

Remarks

- Before you get an integer variable, you must open its form. If you get the variable after opening the form and before reading or replacing the variable on the form, the program returns the initial value specified by the form designer.

- If the form designer specifies data validation rules and error processing to display an error message or form, the program does not need to look at the variable status parameter.

  If the form designer specifies data validation rules and no error processing, the program must look at the variable status parameter.

  If the form designer specifies no data validation rules, the program must look at the variable status parameter.

Getting the Next Event

Purpose FDP$GET_NEXT_EVENT gets the next event resulting from the most recent FDP$READ_FORMS procedure.

Format **FDP$GET_NEXT_EVENT (event_name, event_normal, event_position, last_event, status)**

Parameters event_name: VAR of ost$name;

A data name to receive the application user's event.

event_normal: VAR of boolean;

A data name to receive the event normal indication. If the event is normal, TRUE is returned; if the event is abnormal, FALSE is returned.

event_position: VAR of fdt$event_position;

A data name to receive the position of the event. The character position in the upper left corner of a screen or a form is 1; the x position increases by 1 for each character, counting from left to right; the y position increases by 1 for each character counting from top to bottom.

The following fields are returned:

Field	Meaning
form_identifier	The identifier of the form on which the event occurred.
screen_x_position	Returns the x position of the event on the terminal screen.
screen_y_position	Returns the y position of the event on the terminal screen.
form_x_position	Returns the x position of the event on the form.
form_y_position	Returns the y position of the event on the form.

For the event_position key, one of the following values is returned:

FDC$FORM_EVENT

The event occurred in a form, but not in an object.

FDC$OBJECT_EVENT

The event occurred in an object. It has the following fields:

Field	Meaning
object_name	The object name. If the object doesn't have a name, the field is OSC$NULL_NAME.
object_occurrence	The occurrence of the object. The first or only occurrence is returned as 1.
object_x_position	The x position of the object. The origin is the upper left corner of the form.
object_y_position	The y position of the object. The origin is the upper left corner of the form.
object_definition_key	A variant record that contains one of the following values: FDC$BOX FDC$LINE FDC$CONSTANT_TEXT FDC$CONSTANT_TEXT_BOX FDC$VARIABLE_TEXT FDC$VARIABLE_TEXT_BOX For variable text and variable text boxes, it also returns the character position of the variable as it appears in the program (which is not necessarily how it appears on the form). The first position is 1.

last_event: VAR of boolean;

Indicates whether this is the last event. If this is the last event, the value is TRUE; if this is not the last event, the value is FALSE.

status: VAR of ost$status;

The record that indicates the results of the procedure.

Conditions fde$bad_data_value

Remarks The FDP$READ_FORMS procedure deletes existing events. If the event is normal, Screen Formatting updates the variables in the added and combined forms containing the event. Later, you can request the transfer of these variables to program storage. If the event is abnormal, Screen Formatting does not update or validate variables.

Getting a Real Variable

Purpose FDP$GET_REAL_VARIABLE gets a value the user entered on a form for a real variable and transfers it to the program.

Format **FDP$GET_REAL_VARIABLE (form_identifier, name, occurrence, variable, variable_status, status)**

Parameters **form_identifier**: fdt$form_identifier;

The identifier established when the form was opened.

name: ost$name;

The name of the variable to get. This name was defined when the form was created.

occurrence: fdt$occurrence;

The occurrence of the variable name. Use 1 for the first or only occurrence.

variable: VAR of real;

The value of the real variable that Screen Formatting generates automatically in the form definition record. If you do not want to use the automatically generated variable, include a variable of type real.

variable_status: VAR of fdt$variable_status;

An ordinal that gives you the status of the variable. The following values are possible:

FDC$INDEFINITE

The user entered an indefinite number.

FDC$INVALID_BDP_DATA

The user entered data that does not correspond to the defined data type.

FDC$INVALID_REAL

The user entered data that is not within the range of real numbers defined for the variable.

FDC$LOSS_OF_SIGNIFICANCE

The user entered a number too large to be converted to the defined real or integer program type.

FDC$NO_ERROR

No error occurred on the variable.

FDC$OVERFLOW

The user entered an exponent that is too large.

FDC$UNDERFLOW

The user entered an exponent that is too small.

status: VAR of ost$status;

The record that indicates the results of the procedure.

Conditions

fde$bad_data_value
fde$invalid_form_identifier
fde$invalid_variable_name
fde$no_space_available
fde$system_error
fde$unknown_occurrence
fde$unknown_variable_name

Remarks

- Before you get a real variable, you must open the form on which the user enters the value. If you get the variable after opening the form and before reading or replacing the variable on the form, the program returns the initial value specified by the form designer.

- If the form designer specifies data validation rules and error processing to display an error message or form, your program does not need to look at the variable status parameter.

  If the form designer specifies data validation rules and no error processing, the program must look at the variable status parameter.

  If the form designer specifies no data validation rules, the program must look at the variable status parameter.

Getting a Record

Purpose FDP$GET_RECORD transfers the values of the form record to the program record.

Format **FDP$GET_RECORD (form_identifier, p_work_area, work_area_length, variable_status, status)**

Parameters form_identifier: fdt$form_identifier;

The identifier established when the form was opened.

p_work_area: { output } ^cell;

Pointer to the work area for the form record. When the form is created, Screen Formatting generates the variable definition entries in this record.

work_area_length: fdt$work_area_length;

The number of cells in the work area to be used in transferring the record.

variable_status: VAR of fdt$variable_status;

An ordinal that gives you the status of the variable. The following values are possible:

FDC$INDEFINITE

The user entered an indefinite number.

FDC$INFINITE

The user entered an infinite number.

FDC$INVALID_BDP_DATA

The user entered data that does not correspond to the defined data type.

FDC$INVALID_INTEGER

The user entered data that is not within the range of integer numbers defined for the variable.

FDC$INVALID_REAL

The user entered data that is not within the range of real numbers defined for the variable.

FDC$INVALID_STRING

The user entered data that does not match the strings defined for the variable.

FDC$LOSS_OF_SIGNIFICANCE

The user entered a number too large to be converted to the defined real or integer data type.

FDC$NO_ERROR

No error occurred on the variable.

FDC$OVERFLOW

The user entered an exponent that is too large.

FDC$UNDERFLOW

The user entered an exponent that is too small.

status: VAR of ost$status;

The record that indicates the results of the procedure.

Conditions

ide$bad_data_value
fde$form_has_no_variables
fde$invalid_form_identifier
fde$no_space_available
fde$system_error
fde$work_invalid

Remarks

- Before you get a record for a form, you must open the form. If you get the record after opening the form and before reading or replacing the record, the program returns the initial value specified by the form designer.

- If the form designer specifies data validation rules and error processing to display an error message or form, your program does not need to look at the variable status parameter.

  If the form designer specifies data validation rules and no error processing, the program must look at the variable status parameter.

  If the form designer specifies no data validation rules, the program must look at the variable status parameter.

Getting a String Variable

Purpose

FDP$GET_STRING_VARIABLE gets a value the user entered on a form for a string variable and transfers it to the program.

Format

FDP$GET_STRING_VARIABLE (form_identifier, name, occurrence, variable, variable_status, status)

Parameters

form_identifier: fdt$form_identifier;

The identifier established when the form was opened.

name: ost$name;

The name of the variable to get. The name was defined when the form was created.

occurrence: fdt$occurrence;

The occurrence of the variable name. Use 1 for the first or only occurrence.

variable: VAR of fdt$text;

The variable that Screen Formatting generates automatically in the form definition record. The form definition record defines the variable. If you do not want to use the automatically generated variable, include a variable of the following type (* is the number of characters in the variable):

```
string ( * )
```

variable_status: VAR of fdt$variable_status;

An ordinal that gives you the status of the variable. The following values are possible:

FDC$INVALID_STRING

The user entered data that does not match the strings defined for the variable.

FDC$NO_ERROR

No error occurred on the variable.

FDC$VARIABLE_TRUNCATED

The storage length of the VARIABLE parameter is not long enough.

status: VAR of ost$status;

The record that indicates the results of the procedure.

Conditions

fde$bad_data_value
fde$invalid_form_identifier
fde$invalid_variable_name
fde$no_space_available
fde$system_error
fde$unknown_occurrence
fde$unknown_variable_name
fde$wrong_variable_name

Remarks

- Before you get a string variable, you must open the form on which the user enters the value. If you get the variable after opening the form and before reading or replacing the variable on the form, the program returns the initial value specified by the form designer.
- If the form designer specifies data validation rules and error processing to display an error message or form, your program does not need to look at the variable status parameter.

  If the form designer specifies data validation rules and no error processing, the program must look at the variable status parameter.

  If the form designer specifies no data validation rules, the program must look at the variable status parameter.

Opening a Form

Purpose

FDP$OPEN_FORM locates a form and prepares it for use by the program.

Format

FDP$OPEN_FORM (form_name, form_identifier, status)

Parameters

form_name: ost$name;

The name of the form you want to open.

form_identifier: VAR { input-output } of fdt$form_identifier;

The form identifier established for the form. Other Screen Formatting procedures use this identifier when referencing the form.

status: VAR of ost$status;

The record that indicates the results of the procedure.

Conditions

fde$bad_data_value
fde$form_already_open
fde$form_not_ended
fde$form_requires_conversion
fde$invalid_form_identifier
fde$invalid_form_name
fde$no_space_available
fde$system_error
fde$terminal_not_identified
fde$unknown_form_name

Remarks

- Screen Formatting locates a form as follows:
  - If the form name is blank, Screen Formatting assumes that the form identifier specifies the required dynamically created form.
  - If the form name is not blank, Screen Formatting searches the list of ended dynamically created forms.
  - If the form name is not blank and is not in the list of ended dynamically created forms, Screen Formatting searches the command library list to find the form name on the object libraries. (You specify the order in which Screen Formatting searches the list using the NOS/VE command CREATE_COMMAND_LIST_ENTRY).
- Executing FDP$OPEN_FORM does not display the form on the screen. (See Reading a Form or Showing a Form.)
- The form identifier that FDP$OPEN_FORM returns identifies the instance of open for a form. Forms dynamically created have only one instance of open. Forms stored on object libraries can have more than one instance of open. For each instance of open, Screen Formatting maintains the working environment (current value of variables and their display attributes) of the form.

Popping a Form

Purpose FDP$POP_FORMS deletes forms scheduled (added or combined) since the last FDP$PUSH_FORMS call.

Format **FDP$POP_FORMS (status)**

Parameters status: VAR of ost$status;

The record that indicates the results of the procedure.

Conditions fde$bad_data_value
fde$no_forms_to_pop

Remarks Events associated with the last list of pushed forms become active.

Positioning a Form

Purpose FDP$POSITION_FORM schedules moving a form to a new location. Using this procedure, you can define a form at one location and display it at another location, or you can move a form from where it is currently displayed to a new location.

Format **FDP$POSITION_FORM (form_identifier, screen_x_position, screen_y_position, status)**

Parameters form_identifier: fdt$form_identifier;

The form identifier established when the form was opened.

screen_x_position: fdt$x_position;

The x position on the screen. The character position in the upper left corner of the screen is 1, and the x position increases by 1 for each character counting from left to right.

screen_y_position: fdt$y_position;

The y position on the screen. The character position in the upper left corner of the screen is 1, and the y position increases by 1 for each character counting from top to bottom.

status: VAR of ost$status;

The record that indicates the results of the procedure.

Conditions
fde$bad_data_value
fde$form_pushed
fde$form_too_large_for_screen
fde$invalid_form_identifier
fde$no_space_available
fde$system_error

Remarks

- When the program calls either the FDP$READ_FORMS or FDP$SHOW_FORMS procedure, Screen Formatting displays the form on the screen at the position specified in the call to FDP$POSITION_FORM.
- If you call this procedure while the form is displayed, the form is deleted from its current location and added at the new location. The added form is displayed on top of any other form occupying the same area on the screen.
- If you call this procedure before the form is displayed, the form is displayed at the specified location.
- Before you position a form, you must open it.
- You cannot position a pushed form.

Pushing a Form

Purpose FDP$PUSH_FORMS causes Screen Formatting to record added and combined forms so you can return to them later.

Format **FDP$PUSH_FORMS (status)**

Parameters status: VAR of ost$status;

The record that indicates the results of the procedure.

Conditions fde$bad_data_value
fde$no_forms_to_push

Remarks

- Events associated with these forms are not passed to the program.
- A program cannot change or close a pushed form.
- Pushed forms are displayed on the screen. If you want newly added forms to appear on a blank screen, first add a blank form that covers the screen.

  Updates to the screen continue to show the pushed forms.
- This subroutine deactivates the events associated with forms scheduled for display (added or combined) since the last push call.

Reading a Form

Purpose FDP$READ_FORMS updates the terminal screen and accepts input from the application user.

Format **FDP$READ_FORMS (status)**

Parameters status: VAR of ost$status;

The record that indicates the results of the procedure.

Conditions
fde$bad_data_value
fde$no_events_active
fde$no_forms_to_read
fde$system_error
fde$terminal_disconnected

Remarks

- A call to FDP$READ_FORMS:
  - Displays all the forms you scheduled for display and have not deleted. If you added or combined forms since the last FDP$READ_FORMS or FDP$SHOW_FORMS call, it displays them for the first time.
  - Removes from the screen the forms you deleted since the last FDP$READ_FORMS or FDP$SHOW_FORMS call.
  - Updates on the screen the variables replaced since the last FDP$READ_FORMS or FDP$SHOW_FORMS call.
  - Updates on the screen the objects for which display attributes were set or reset since the last FDP$READ_FORMS or FDP$SHOW_FORMS call.
- Events not retrieved with the FDP$GET_NEXT_EVENT procedure are deleted before any input is accepted from the user.

- The FDP$READ_FORMS procedure does not execute unless the forms scheduled for display contain at least one active event.

- After issuing this request, your program does not regain control until the user issues a normal event and Screen Formatting validates all the data, or the user issues an abnormal event.

Replacing an Integer Variable

Purpose FDP$REPLACE_INTEGER_VARIABLE transfers a program variable to Screen Formatting.

Format **FDP$REPLACE_INTEGER_VARIABLE (form_identifier, name, occurrence, variable, variable_status, status)**

Parameters **form_identifier**: fdt$form_identifier;

The identifier established when the form was opened.

name: ost$name;

The name of the variable to replace. This name was defined when the form was created.

occurrence: fdt$occurrence;

The occurrence of the variable name. Use 1 for the first or only occurrence.

variable: integer;

The integer variable that Screen Formatting generates automatically in the form definition record. If you do not want to use the automatically generated variable, include a variable of type integer.

variable_status: VAR of fdt$variable_status;

An ordinal that gives you the status of the variable. The following values are possible:

FDC$LOSS_OF_SIGNIFICANCE

The program supplied a value that is too large for the form field.

FDC$NO_ERROR

No error occurred on the variable.

FDC$OUTPUT_FORMAT_BAD

The output format defined for the variable cannot output the variable.

status: VAR of ost$status;

The record that indicates the results of the procedure.

Conditions

fde$bad_data_value
fde$form_pushed
fde$invalid_form_identifier
fde$invalid_variable_name
fde$no_space_available
fde$system_error
fde$unknown_occurrence
fde$unknown_variable_name
fde$wrong_variable_type

Remarks

- When you call either the FDP$READ_FORMS or FDP$SHOW_FORMS procedure, Screen Formatting replaces the integer variable on the terminal screen.
- Before you replace an integer variable, you must open the form on which it is replaced.
- You cannot replace an integer variable for a pushed form.
- If the integer variable is not valid, it is not replaced.

Replacing a Real Variable

Purpose FDP$REPLACE_REAL_VARIABLE transfers a real program variable to Screen Formatting.

Format **FDP$REPLACE_REAL_VARIABLE (form_identifier, name, occurrence, variable, variable_status, status)**

Parameters **form_identifier**: fdt$form_identifier;

The identifier established when the form was opened.

name: ost$name;

The name of the variable to replace. This name was defined when the form was created.

occurrence: fdt$occurrence;

The occurrence of the variable name. Use 1 for the first or only occurrence.

variable: real;

The value of the real variable that Screen Formatting generates automatically in the form definition record. If you do not want to use the automatically generated variable, include a variable of type real.

variable_status: VAR of variable_status;

An ordinal that gives you the status of the variable. The following values are possible:

FDC$LOSS_OF_SIGNIFICANCE

The value the program supplied is too large for the form variable.

FDC$NO_ERROR

No error occurred on the variable.

FDC$OUTPUT_FORMAT_BAD

The output format defined for the variable cannot output the variable.

status: VAR of status;

The record that indicates the results of the procedure.

Conditions

fde$bad_data_value
fde$form_pushed
fde$invalid_form_identifier
fde$no_space_available
fde$system_error
fde$unknown_occurrence
fde$unknown_variable_name
fde$variable_name
fde$wrong_variable_type

Remarks

- When you call either the FDP$READ_FORMS or FDP$SHOW_FORMS procedure, Screen Formatting replaces the real variable on the terminal screen.
- Before you replace a real variable, you must open the form on which it is replaced.
- You cannot replace a real variable for a pushed form.
- If the real variable is not valid, it is not replaced.

Replacing a Record

Purpose

FDP$REPLACE_RECORD transfers values of program variables to Screen Formatting for later display on a form.

Format

FDP$REPLACE_RECORD (form_identifier, p_work_area, work_area_length, variable_status, status)

Parameters

form_identifier: fdt$form_identifier;

The identifier established when the form was opened.

p_work_area: ^cell;

The pointer to the program work area for variables. When the form is created, Screen Formatting generates a type definition for you to assign to this variable.

work_area_length: fdt$work_area_length;

The number of cells in the work area.

variable_status: VAR of fdt$variable_status;

An ordinal that gives you the status of the variables. The following values are possible:

FDC$INDEFINITE

The program supplied an indefinite number.

FDC$INFINITE

The program supplied an infinite number.

FDC$LOSS_OF_SIGNIFICANCE

The program supplied a number that is too large to be converted to the form variable size.

FDC$NO_ERROR

No error occurred on the variables.

FDC$OUTPUT_FORMAT_BAD

The output format defined for a variable cannot output the variable.

FDC$OVERFLOW

The program supplied an exponent that is too large.

FDC$UNDERFLOW

The program supplied an exponent that is too small.

status: VAR of ost$status;

The record that indicates the results of the procedure.

Conditions

fde$bad_data_value
fde$form_has_no_variables
fde$form_pushed
fde$invalid_form_identifier
fde$no_space_available
fde$work_invalid

Remarks

- When the program calls either the FDP$READ_FORMS or FDP$SHOW_FORMS procedure, Screen Formatting replaces the variables on the terminal screen with the values stored in Screen Formatting.

- Before you replace a record, you must open the form on which the variables are replaced.

- You cannot replace a record for a pushed form.

Replacing a String Variable

Purpose FDP$REPLACE_STRING_VARIABLE transfers a program string variable to Screen Formatting.

Format **FDP$REPLACE_STRING_VARIABLE (form_identifier, name, occurrence, variable, variable_status, status)**

Parameters form_identifier: fdt$form_identifier;

The identifier established when the form was opened.

name: ost$name;

The name of the variable to replace. This name was defined when the form was created.

occurrence: fdt$occurrence;

The occurrence of the variable name. Use 1 for the first or only occurrence.

variable: fdt$text;

The variable that Screen Formatting generates automatically in the form definition record. The form definition record defines the variable. If you do not want to use the automatically generated variable, use a variable of the following type (* is the number of characters in the variable):

string (*)

variable_status: VAR of fdt$variable_status;

An ordinal that gives you the status of the variable. The following value is possible:

FDC$NO_ERROR

No error occurred on the variable.

status: VAR of ost$status;

The record that indicates the results of the procedure.

Conditions

fde$bad_data_value
fde$form_pushed
fde$invalid_form_identifier
fde$invalid_variable_name
fde$no_space_available
fde$system_error
fde$unknown_occurrence
fde$unknown_variable_name
fde$wrong_variable_type

Remarks

- When the program calls either the FDP$READ_FORMS or FDP$SHOW_FORMS procedure, Screen Formatting replaces the string variable on the terminal screen.
- Before you replace a string variable, you must open the form on which it is replaced.
- You cannot replace a string variable for a pushed form.
- If the string variable is not valid, it is not replaced.
- If the form specifies that the data must be in upper case, Screen Formatting converts it to upper case before storing the data in the form.

Resetting a Form

Purpose FDP$RESET_FORM resets the form to the state specified by the form definition.

Format **FDP$RESET_FORM (form_identifier, status)**

Parameters form_identifier: fdt$form_identifier;

The identifier established when the form was opened.

status: VAR of ost$status;

The record that indicates the results of the procedure.

Conditions

fde$bad_data_value
fde$form_pushed
fde$invalid_form_identifier
fde$no_space_available
fde$system_error

Remarks

- When the program calls either the FDP$READ_FORMS or FDP$SHOW_FORMS procedure, Screen Formatting displays the form on the terminal screen with the reset specifications.
- All variables belonging to the form have their initial values and display attributes. The form is in its defined position.
- Before you reset a form, you must open it.
- You cannot reset a pushed form.

Resetting an Object Attribute

Purpose FDP$RESET_OBJECT_ATTRIBUTE resets the display attributes for an object to those specified in the form definition.

Format **FDP$RESET_OBJECT_ATTRIBUTE (form_identifier, object_name, occurrence, status)**

Parameters form_identifier: fdt$form_identifier;

The identifier established when the form was opened.

object_name: ost$name;

The name of the object whose attributes are being reset. This name was defined when the form was created.

occurrence: fdt$occurrence;

The occurrence of the object. For the first or only occurrence, use 1.

status: VAR of ost$status;

The record that indicates the results of the procedure.

Conditions

fde$bad_data_value
fde$form_not_scheduled
fde$form_pushed
fde$invalid_form_identifier
fde$invalid_object_name
fde$invalid_occurrence
fde$no_space_available
fde$unknown_object_name

Remarks

- You can reset the attributes of objects that are variable text, constant text, lines, or boxes.
- Before you reset the attribute of an object, you must open and either add or combine the form the object is on.
- When the program calls either the FDP$READ_FORMS or FDP$SHOW_FORMS procedure, Screen Formatting displays the object using the reset attributes.

Setting the Cursor Position

Purpose FDP$SET_CURSOR_POSITION sets the cursor to a selected position for later display.

Format **FDP$SET_CURSOR_POSITION (form_identifier, object_name, occurrence, character_position, status)**

Parameters form_identifier: fdt$form_identifier;

The identifier established when the form was opened.

object_name: ost$name;

The name of the object on which you want to set the cursor. This name was defined when the form was created.

occurrence: fdt$occurrence;

The integer specifying the occurrence of the object name. Use 1 for the first occurrence.

character_position: fdt$character_position;

The character position to which you want to set the cursor. Use 1 for the first character position.

status: VAR of ost$status;

The record that indicates the results of the procedure.

Conditions

fde$bad_data_value
fde$form_not_scheduled
fde$form_pushed
fde$invalid_character_position
fde$invalid_form_identifier
fde$invalid_object_name
fde$no_object_available_defined
fde$no_space_available
fde$system_error
fde$unknown_object_name
fde$unknown_occurrence

Remarks

- One use of this procedure is to alter the default sequence of the application user's entry of variables. In the default sequence, Screen Formatting places the cursor on the first input variable of the highest priority form. The highest priority form is the form last added, combined, or positioned.

  At terminals with protected fields, the user tabs from one variable text object to the next. The cursor starts at the top line of the form; it moves from left to right on each line. When no variable text object appears on a line, the cursor moves down to the next line. At terminals without protected fields, the user must move the cursor using the arrow keys or the tab and return keys.

- When you call either the FDP$READ_FORMS or FDP$SHOW_FORMS procedure, Screen Formatting updates the terminal screen with the cursor at the specified position.

- If the position you specify is not visible on the screen, Screen Formatting shifts the data to make the cursor visible.

- The cursor position is in effect only for the next screen update from reading or showing forms.

- Before you set the cursor position on a form, you must open the form and either add or combine it.

- You cannot set the cursor position in a pushed form.

Setting Line Mode

Purpose FDP$SET_LINE_MODE begins line-by-line interaction with an application user.

Format **FDP$SET_LINE_MODE (status)**

Parameters status: VAR of ost$status;

The record that indicates the results of the procedure.

Conditions fde$bad_data_value

Remarks

- Use this call for extended dialogues in line mode. For short dialogues, Screen Formatting automatically switches to the proper mode (line or screen) but resources used for screen mode interaction remain.
- This call releases all screen mode resources:
  - Open forms are closed.
  - The mode is set to line.

Setting an Object Attribute

Purpose FDP$SET_OBJECT_ATTRIBUTE changes a display attribute for an object.

Format **FDP$SET_OBJECT_ATTRIBUTE (form_identifier, object_name, occurrence, attribute, status)**

Parameters form_identifier: fdt$form_identifier;

The identifier established when the form was opened.

object_name: ost$name;

The name of the object whose display attribute is being reset.

occurrence: fdt$occurrence;

The occurrence of the object. For the first or only occurrence, use 1.

attribute: ost$name;

The name given to the display attribute when it was defined on the form. The attribute used here is defined for the form and not for a specific object. When using Screen Design Facility, screen attributes are defined through the ATTRIB function. When using a CYBIL program, the ADD_DISPLAY_DEFINITION attribute record defines form attributes.

status: VAR of ost$status;

The record that indicates the results of the procedure.

Conditions fde$bad_data_value
fde$form_not_scheduled
fde$form_pushed
fde$invalid_attribute_position
fde$invalid_form_identifier
fde$invalid_object_name
fde$invalid_occurrence
fde$no_space_available
fde$unknown_display_name
fde$unknown_object_name
fde$unknown_occurrence

Remarks

- You can set the attributes of objects that are variable text, constant text, lines, or boxes.
- Changed attributes replace existing attributes.
- When you call either the FDP$READ_FORMS or FDP$SHOW_FORMS procedure, Screen Formatting displays the object using the set attributes.
- If the object you specify is not visible on the screen, Screen Formatting shifts the data to make the object visible.
- Before you set the attribute of an object, you must open the form the object is on and either add or combine it.
- You cannot set attributes of objects on a pushed form.

Showing Forms

Purpose FDP$SHOW_FORMS updates the terminal screen.

Format **FDP$SHOW_FORMS (status)**

Parameters status: VAR of ost$status;

A record that indicates the results of the procedure.

Conditions
fde$bad_data_value
fde$form_too_large_for_screen
fde$form_to_show
fde$no_space_available
fde$system_error
fde$terminal_disconnected

Remarks

- When none of the forms scheduled for display has an event or input variable defined, use this procedure instead of FDP$READ_FORMS.

- When you do not want any input from the terminal user, use this subroutine.

- A call to FDP$SHOW_FORMS:

  - Displays all the forms you have scheduled for display and have not deleted. If you added or combined forms since the last FDP$READ_FORMS or FDP$SHOW_FORMS call, it displays them for the first time.

  - Removes from the screen the forms you deleted since the last FDP$READ_FORMS or FDP$SHOW_FORMS call.

  - Displays variables replaced since the last FDP$READ_FORMS or FDP$SHOW_FORMS call.

  - Displays objects with attributes set or reset since the last FDP$READ_FORMS or FDP$SHOW_FORMS call.

Using CYBIL to Create Forms 7

More About Forms 7-2
Constant Text Objects 7-3
Variable Text Objects 7-3
Data Flow Attributes 7-4
Data Type Attributes 7-4
Output Formatting Attributes 7-5
Data Validation Attributes 7-6
Tables 7-8
Graphic Objects 7-9
Events 7-9
Defining Screen Formatting Tasks for Events 7-10
Standard Events 7-15
Display Attributes 7-15
Error and Help Information 7-16
Creating Unique Forms for Error and Help Information 7-17
Using the Default Form for Error and Help Information 7-17

How to Create a Form 7-19

The Design Specification 7-20

Instructions for Designing Forms 7-22
Designing a Form Dynamically 7-22
Designing a Form Interactively 7-25

Rectangle Form Program 7-34

Creating Form Definition Records for Existing Forms 7-42

Attributes for a Form 7-43
Basic Form Attributes 7-44
Creating and Changing Forms 7-44
Getting Basic Form Attributes 7-56
Variable Attributes 7-60
Creating and Changing Variables 7-60
Getting Variable Attributes 7-69
Table Attributes 7-73
Creating and Changing Tables 7-73
Getting Table Attributes 7-75
Form Definition Record Attributes 7-76
Changing Record Attributes 7-76
Getting Record Attributes 7-77

Object Attributes 7-78
Creating and Changing Objects 7-78
Getting Object Attributes 7-81

CYBIL Screen Formatting Procedures 7-85
Changing a Form 7-86
Changing the Form Definition Record 7-87
Changing an Object 7-88
Changing a Stored Object 7-89
Changing a Table 7-90
Changing a Variable 7-91
Converting to Program Variable 7-92
Converting to Screen Variable 7-94
Copying an Area 7-96
Copying a Form 7-98
Creating Constant Text 7-99
Creating a Design Form 7-100
Creating Design Text 7-102
Creating a Form 7-103
Creating an Event Form 7-104
Creating a Mark 7-106
Creating an Object 7-108
Creating a Stored Object 7-113
Creating a Table 7-115
Creating a Variable 7-116
Deleting an Area 7-118
Deleting a Mark 7-119
Deleting an Object 7-120
Deleting a Stored Object 7-121
Deleting a Table 7-122
Deleting a Variable 7-123
Editing a Form 7-124
Ending a Form 7-125
Getting Form Attributes 7-126
Getting Form Names 7-127
Getting Form Objects 7-128
Getting Object Attributes 7-130
Getting Record Attributes 7-131
Getting a Stored Object 7-132
Getting Table Attributes 7-133
Getting Variable Attributes 7-134
Moving an Area 7-135
Writing a Form Definition 7-137
Writing a Form Definition Record 7-138

Using CYBIL to Create Forms 7

Chapter 1 presented an example of creating and managing forms. It demonstrated that both the designer and the programmer have specific tasks to accomplish. When creating and managing the forms using a COBOL, FORTRAN, Pascal, or CYBIL program, the following tasks need to be accomplished:

1. The form designer and programmer plan the forms and program.

2. The form designer creates the forms specifying the language that will be used to display the forms as the form processor (or programming language) and prepares a design specification.

3. The form designer puts the forms in an object library and makes the form definition record available to the programmer. Each record defines the variables on a particular form and is written in language specified when the form was created.

4. The programmer codes the program, including calls to Screen Formatting based on the design specification. These calls manage the forms created by the designer.

5. The programmer expands and compiles the program.

6. The programmer writes a user procedure to start the application and helps the user set up the correct terminal environment for using the forms.

When the last task is complete, the program and forms are ready for the application user.

This chapter expands on the introduction of forms given in chapter 1 and explains how to create and change forms using CYBIL procedures with Screen Formatting. At the end of the chapter, the formats and parameters are described for each CYBIL procedure you can use.

More About Forms

As presented in chapter 1, a *form* is an organized collection of objects (visual clues, entry fields, prompts or other text) which are treated as a unit on the screen.

Forms have the following general properties:

- A form occupies a rectangular area on the screen, either the entire screen or a part of it.
- Two or more forms can be displayed simultaneously.
- All forms are opaque. When one form covers another form, the covered form is not visible.
- Forms have a priority for display on the terminal. The current form covers those previously displayed.
- The lifetime of a form cannot exceed the lifetime of the program displaying the form.

A form can contain the following:

- Constant text objects (protected text, such as titles or labels)
- Variable text objects (unprotected text, such as user or program data entry)
- Tables (occurrences of variables)
- Graphic objects (lines or boxes)
- Events (actions the user executes, such as pressing function keys)
- Display attributes (inverse video, color)
- Error and help information (messages or forms)

The following sections describe in detail what a form can contain.

Constant Text Objects

A constant text object is text you do not intend the user or program to change, such as a title or a label. Constant text objects have the following properties:

- Display attributes can be associated with them.
- They are not transferred between programs and forms.
- They can occupy part or all of one or more lines on the form.
- They can be formatted differently from line to line.
- If the user temporarily changes them, the text is reset to its initial value as soon as possible.
- They can have names and occurrences, however names and occurrences are not required. When you define occurrences, the object does not need to belong to a table.

Variable Text Objects

Variable text objects are areas where data is entered by the user or the program. They have the following properties:

- A program refers to them by a variable name.
- When a variable name occurs more than once on a form, the variable text objects must be part of a table.
- They can occupy part or all of one or more lines on the form.
- You can specify the following attributes for variables:
  - Data flow
  - Data type
  - Output formatting
  - Data validation

  These attributes are described in the following sections.

Data Flow Attributes

For variable text objects, you can specify how you want the data to flow to and from the user and the program.

The modes are:

- User Input Only

  When the user enters data, Screen Formatting attempts to prevent the data from being echoed to the screen. If the terminal does not support this mode, the data is replaced with blanks as soon as possible.

- Output Only

  Screen Formatting attempts to prevent new data from being entered. If the terminal does not protect text, the form is restored to its correct value after it has been changed.

- Terminal Input and Output

  When the user enters data, it appears on the screen. A program can change this data.

- Program Input and Output

  The data does not appear on the screen. A program uses a variable to record information about the user's interaction. This is called hidden text.

Data Type Attributes

For more convenient program processing, you can convert the type of data entered by the user as follows:

- Character

  The program receives data as entered by the user (no conversion).

- Uppercase Character

  The user's input is converted to uppercase and passed to the program. When the program passes data to the screen, it is also converted to uppercase.

- Integer

  The user's input is converted to integers and passed to the program.

- Real

  The user's input is converted to real numbers and passed to the program.

Output Formatting Attributes

You specify the following output formatting attributes:

- You can specify that the output has a currency format.
- You can assign an initial display attribute to variable text objects. (A program can temporarily change the attribute and later reset the attribute to its initial value.)
- When text is on more than one line, you define the text box by specifying its location, height, and width.

  You also specify how the text is mapped into the text box as follows:

  - Wrap Characters

    Text that does not fit on a line is placed on the next line. Data that exceeds the text box area is not displayed (the user can scroll to the undisplayed text).

  - Wrap Words

    When possible, text is displayed in its box so that words are not broken between lines (a space indicates the end of a word). Data that exceeds the text box area is not displayed (the user can scroll to the undisplayed text).

Data Validation Attributes

User-entered data is automatically validated against a set of application-defined rules. The rules typically specify the format and values for the data. You specify the application rules when creating the form.

To provide a smooth interface for users when they encounter difficulties in using a form, you can do the following:

- Create help messages and forms.

  The message or form can be associated with the entire form or a specific variable text object on the form.

- Create error messages and forms.

  A message or form you create is automatically displayed when an error is detected.

- Change the highlighting display attribute for errors.

  Errors are automatically highlighted in inverse video. You can change the highlighting to another display attribute.

- Allow users to move to another part of the program without correcting an invalid value.

  You define abnormal events that return to the program without storing the values entered by the user.

- Allow users to enter just enough characters to make a text string unique so the system recognizes which valid character string it represents.

  When defining the values for a variable, you specify the strings that are acceptable entries and whether or not the system will recognize unique substrings.

To provide additional help to the application programmer in validating data, you can define data according to a specific format and content.

You can define the format to allow numbers in the following:

- FORTRAN integer formats.

  The integer format includes only numeric characters (0 through 9) or signed numeric characters.

- FORTRAN real formats.

  FORTRAN programmers know these as: Fw.d, Ew.d, Ew.dEe, Gw.d, and Gw.dEe edit descriptors.

You can define the content to allow the following:

- Any characters.
- Only alphabetic characters (A through Z; a through z).
- One or more integer ranges.
- One or more real ranges.
- Unique substrings that contain enough characters to identify valid strings the system can recognize.
- Only valid real or integer numbers.

  When the conversion of user input to program variables results in loss of significance or overflow, the converted data is invalid.

Tables

A table consists of one or more occurrences of one or more variable text objects. These objects can appear anywhere on the form. Each field on the form belonging to the object represents an occurrence in the table.

Use a table to group variable text objects in the following ways:

- Objects whose attributes are identical except for their position on the form.

  For an example definition of a table with two occurrences representing the sides of a rectangle, refer to the program segment labeled

  ```
  { Create table of rectangle sides
  ```

  in the *Rectangle Form Program* later in this chapter. For a representation of the Rectangle Form defined by this program, refer to Figure 4-2.

- Objects that are logically related, such as quantity, part number, description, and cost on an order form.

  For example, the following picture shows a table on a form in an inventory program:

  ```
  COMPUTER PARTS INVENTORY:

  Part Number                    Description

  ___________      ________________________________________
  ___________      ________________________________________
  ___________      ________________________________________
  ___________      ________________________________________
  ```

  The picture shows four separate lines for part numbers and descriptions. The table represented by the lines consists of two variables. One variable contains the part number and the other variable contains the description. Each line represents an occurrence in the table.

The number of occurrences in a table can affect performance. We recommend that you have no more than 200 occurrences in any table.

Graphic Objects

Graphic objects include boxes and line drawings (note that some terminals support only vertical and horizontal lines). A box can be drawn around a form or within it.

Graphic objects have the following general properties:

- They can include display attributes, such as line thickness.
- A program can change the display attribute.
- Object names can be assigned to them.
- They cannot intersect one another.
- They are protected (their text cannot be accidently changed).

Events

An *event* is an action the user executes to return control to Screen Formatting, such as pressing a function key or the return key. Events are defined by specifying both the *event trigger* that identifies the set of keystrokes the user makes and the *event action* that tells Screen Formatting to either perform a task itself or pass it through to the program. The following are examples of events:

- When the return key is pressed, control is passed to the program in order to display the next form.
- When the keys that perform the *move forward* event are pressed, a table is paged forward. (The standard events defined by Control Data are described later in this section.) The event is not passed through to the program.

You can allow the following combinations:

- The user enters data and then executes an event.
- The user places the cursor on an object and then executes an event.
- The user executes an event with no prior action.

When there are two or more forms on the screen, the interpretation of events depends on whether the forms were added or combined before they were displayed. If the forms were added, Screen Formatting

processes only events associated with the form the user places the cursor on. Even though the events displayed relate to all the forms appearing on the screen, when the events are processed the only data affected is that appearing on the form containing the cursor.

If a form is combined with another form, the events associated with the first form are for the combined form also. Data appearing on either form is affected when an event is processed.

In the design specifications, identify whether forms should be added or combined before they are displayed. For a description of the FDP$ADD_FORM and FDP$COMBINE_FORM procedures, refer to chapter 6, Using CYBIL to Manage Forms.

The following sections describe the tasks you can specify and the standard events you can use.

Defining Screen Formatting Tasks for Events

For each event, you must specify one of the following tasks:

- Make a normal return to the program.
- Make an abnormal return to the program.
- Page or scroll on the form.
- Display help forms.
- Erase error and help forms.
- Move cursor to the next or previous input variable.

The tasks are described individually in the next sections.

Normal Return to Program

When you want the application to process data the user enters on the form, define an event to make a normal return to the program. For more information on normal versus abnormal events, refer to *Processing Events and Data* in chapter 6, Using CYBIL to Manage Forms.

Before returning control to the program, Screen Formatting uses the data definitions to validate all the values entered by the user. For each invalid value, it does the following:

1. Highlights the invalid value.

   For each defined variable, select the display attribute for highlighting errors. The default display attribute for errors is inverse video.

2. Sets the cursor to the first character of the invalid value.

3. Displays a message that explains how to correct the error.

   Define the error message either on its own form or in the program. If an error message is not defined, this step is skipped.

4. Waits for the user to reenter the value.

Each time the user executes a normal event, the validation process is repeated until no invalid values are left or until the user executes an abnormal event. The program does not regain control until one of these two occurs. Each time Screen Formatting checks for valid values, it removes previous error highlights and error messages.

Abnormal Return to Program

When you want the application to perform a task other than processing user-entered data, define an event to make an abnormal return to the program.

Examples of program tasks that require definition of an abnormal return are:

- Quitting the program.
- Displaying a different form without checking for valid data on the first form.

For an example definition of an abnormal event, refer to the program segment labeled

```
{ Define abnormal events
```

in the *Rectangle Form Program* later in this chapter.

With several abnormal returns defined (such as Quit, Help, and Back), the user has more flexibility in telling the program what to do next.

When the user makes an abnormal return to the program, Screen Formatting does not update or validate variables for the application.

Paging and Scrolling

When you have more data for a variable or table than you can display at one time, define an event to page or scroll through the data.

Paging involves displaying the next or previous group of data. To page when there is more than one table or variable that uses paging defined on a form, the user must position the cursor on the table or variable to be paged.

Scrolling moves the character containing the cursor to either the top or the bottom of the table or variable. The rest of the data is then realigned.

For each form, you can define only one event to page forward, one event to page backward, one event to scroll forward, and one event to scroll backward. Screen Formatting performs paging and scrolling tasks; the application program does not need to execute any statements, nor does it regain control.

Displaying Help

When you define help information for a form, you must also define an event that displays the help. Use the standard *request help* event defined by Control Data. The event action is FDC$DISPLAY_HELP, described later in this chapter under *Basic Form Attributes*. Screen Formatting performs this task; the application program does not need to execute any statements, nor does it regain control.

The help message displayed depends on the position of the cursor:

- When the user positions the cursor on a variable text object and executes the *request help* event, the help message for the variable is displayed.

- When the user positions the cursor anywhere else on the form and executes the *request help* event, the help message for the entire form is displayed.

Only one help or error message can appear on the screen at a time.

Executing a normal or abnormal event erases help messages. You can also define an event to erase the help message without returning to the program.

If you did not define help information when you created the form, a message explaining that the key has no definition is displayed when a user executes the *request help* event.

Erasing Error Message and Help Forms

Because an error form or help form may cover other forms on the screen, the user may want to erase them. To erase help forms, define an event that does this. Specify the trigger and event action for erasing help forms as FDC$ERASE_HELP. This event also erases error message forms. By positioning the cursor inside the error or help form and pressing the keys assigned, the user can remove the form from the screen before returning to the program.

Screen Formatting also supplies the standard *back to previous context* event to erase error message or help forms. The user positions the cursor in the error form and executes the *back to previous context* event.

A normal or abnormal event also erases error and help forms.

Moving the Cursor to the Next or Previous Input Variable

You can define events to move the cursor to the next input variable on a form and to the previous input variable on a form. These types of events will be helpful in the following circumstances:

- When the application user's terminal does not have protected fields. For most terminals, the application user moves from one input variable to the next or to the previous variable by pressing tab keys. At a terminal that does not have protected fields, the user must position the cursor to the next input variable or to the previous one using the arrow keys or the tab and return keys. Pressing an event to position the cursor is usually faster at this type of terminal.

- When the application displays more than one form at the same time. Executing an event that positions the cursor at the next input variable keeps the cursor in the same form. When a user presses a tab key, the cursor goes to the next input variable on the screen. This variable may not be on the same form when more than one form appears at the same time.

Standard Events

To be consistent with other NOS/VE screen applications, use the following Control Data-defined standard events in your own applications:

Standard Event	Description
Move backward	Display the previous set of data
Move to first	Display the first set of data
Move forward	Display the next set of data
Move to last	Display the last set of data
Back to previous context	Switch to a previously shown display
Request help	Display help
Undo last event	Remove changes made to the last event
Redo last event	Restore an undone user event
Quit save	Terminate the application and save any changed data
Alternate exit	Terminate the application and do not save any changed data

Display Attributes

When creating a form, you can associate display attributes with the entire form or with its individual objects. For example, you can specify terminal attributes (such as color, inverse video, and bold lines) or program attributes (such as error, warning, and title).

Foreground and background colors can also be specified for the form. If you don't specify any attributes for an object on a form, the foreground and background colors of the form are used.

For information about creating terminal attributes, refer to the NOS/VE Terminal Definition manual and to appendix C of this manual.

Protected and Unprotected Text

One of the form attributes is text-protection (FDC$PROTECT), which prevents the user from changing the text. The following areas on a form are always protected:

- An area that contains no defined objects.
- Constant text objects.
- Graphic objects.

You define whether or not variable text objects are protected.

Error and Help Information

For each form, you can define the following:

- Error information for each variable text object on the form. The information is displayed when the user enters an invalid value.
- Help information for both the entire form and for each variable text object on the form. The help information is displayed when the user executes the *request help* event (see Events earlier in this chapter).

Error and help information forms are displayed on top of the form currently being used.

You have a choice of two methods for creating an error or help form:

- You can use a form already created by Screen Formatting for this purpose (the default message form) and simply define in your program the message you want to appear.
- You can create your own unique form just as you do any other form.

The default form does not change and always appears in the same place. Because it is small, Screen Formatting automatically provides the paging and scrolling events.

The following sections describe the two methods for creating error and help forms.

Creating Unique Forms for Error and Help Information

As with other forms, the object library on which the error or help forms reside must be in the user's command list. Display the error or help form by including its name in attributes for the user form, as follows:

- For an error form, specify its name on a field in FDC$VARIABLE_ERROR.

  The error form is displayed on the screen after the user enters data that is not valid for a given variable text object.

- For a help form, specify its name on a field in either FDC$FORM_HELP or FDC$VARIABLE_HELP.

  The help form is displayed when the user executes the *request help* event defined for the form.

Using the Default Form for Error and Help Information

Specify the text for the error or help message in the program that creates the user form, and include as a form attribute a pointer to the message:

- For an error message, specify the pointer on a field in FDC$VARIABLE_ERROR.

  An error message is displayed on the screen after a user enters data that is not valid for a given variable text object.

- For a help message, specify the pointer on a field in either FDC$FORM_HELP or FDC$VARIABLE_HELP.

  A help message is displayed when the user executes a *request help* event that has been defined for the form.

The resulting form has the following characteristics:

- It occupies 78 columns and 3 lines.
- It has a box outline.
- The upper left corner of the form is at column 2, row 1 of the screen.
- One variable text object is defined in the form for displaying a message. The variable starts at column 3 of the form (column 4 of the screen). The length of the variable can be up to 255 characters. However, only 76 characters are visible on one line at one time.
- The standard events of *move forward, move backward, back to previous context, move to last,* and *move to first* are defined for the form. The user executes the *back to previous context* event to delete the form. The *move forward* and *move backward* events allow the user to scroll through the message when it is longer than one line. The *move to last* and *move to first* events display the first and last characters of the message.

The online Examples manual has an example named CHANGE_SELECT_FORM that adds help to an existing form using the default form.

You can change the default form by creating your own message form with the name FDM$MESSAGE_FORM (given by FDC$MESSAGE_FORM_NAME). This is the name of the default message form that resides on $SYSTEM.FDF$LIBRARY. Place your form in an object library that is accessible to the application user (most likely the same library containing the forms for the application). To access the new message form, users add the object library to their command list. Then, whenever an error message form is needed, the new form is automatically used.

How to Create a Form

There are two ways to create a form: by means of Screen Design Facility[1] or with Screen Formatting procedures. The latter method requires writing a CYBIL program that uses the procedures documented in this chapter.

With either method, you create a form by defining its attributes. These attributes are placed in a form definition record and stored in an object library. From this record, the program interacts with the form. The following items define a form:

- The size and location of the form on the screen.
- The display attributes that affect the entire form (such as background color).
- Events.
- The program processor that accesses the form (COBOL, FORTRAN, Pascal, SCL, or CYBIL). This processor determines the rules for valid names of variables and tables, and how the record definition is generated.
- Objects on the form such as text, lines, or boxes.
- Display attributes for objects.
- A name for each variable object (so that each can be used by a program without regard to its position on the form).
- Variable attributes.
- Initial cursor position. By default the cursor is positioned at the top-most variable object on the form.

1. For more information, refer to the NOS/VE Screen Design Facility manual.

When creating a form with Screen Formatting procedures in a CYBIL program, you can also:

- Copy a form definition.
- Get or change the definitions for a form, table, variable, or object.
- Delete a table, variable, or object.
- Create error and help messages.

The Design Specification

Once you have created the forms for an application, you should document for the application programmer what you have defined on each form and how the forms interact with each other. The document you create is the design specification for the application. During the planning stage of the forms you no doubt maintained this information in an informal manner. When you have finished creating the forms, give the design specifications and the name of the object library containing the forms to the application programmer writing the managing forms program.

The list that follows contains the necessary information for the CYBIL Rectangle Form created later in this chapter. This information becomes part of the design specification for the example application described in chapter 6, Using CYBIL to Manage Forms. The complete design specification is shown with example program for managing forms in that chapter.

- The name of the form is:

  CYBIL_RECTANGLE_FORM

- The user calls the Rectangle Form from the Select Form.

- The following variable text objects are defined on the form:

SIDE_TABLE	Table that holds values for the rectangle's sides.
SIDE	Areas (two) for user input of values for the rectangle's sides.
AREA	Area for returning value of computed area.
MESSAGE	Area for displaying error messages.

- The following events are defined on the form:

Event	Description
COMPUTE	A normal program event that processes data the user entered on the form. For the Rectangle Form, COMPUTE calculates the area and redisplays the form.
BACK	An abnormal program event that takes the user back to a previous environment. For the Rectangle Form, BACK returns the user to Select Form.
QUIT	An abnormal program event that stops the program.

For instructions on creating a form with Screen Formatting procedures, continue with the next section, Instructions for Designing Forms.

Instructions for Designing Forms

There are two ways of using Screen Formatting to create and change forms: dynamically or interactively.[2] These methods are described in the following sections.

Designing a Form Dynamically

With this method, the program you write creates the form on its own without asking for the user's preferences. This is the simpler of the two methods. If you need to involve the user in the creation of the form, refer to Designing a Form Interactively later in this chapter.

Creating a Form

The following are the steps for dynamically creating a form.

1. Create the form by executing the FDP$CREATE_FORM procedure.

2. Create objects (such as line or box graphics and constant or variable text) by executing the FDP$CREATE_OBJECT procedure. An object can have a name attribute, which allows you to associate a variable definition with the object. You also can change the attributes of an object by referring to the object name.

   The top left corner of the form is the origin of the form coordinate system. The x position starts at 1 and increases by 1 for each character counting from left to right. The y position starts at 1 and increases by 1 for each line counting from top to bottom.

   A variable can be created before or after the creation of the object. Each variable and visible variable table occurrence must have an associated object created before the FDP$END_FORM procedure is issued. An initial value for variable text is specified by using the FDP$CREATE_OBJECT procedure. The value is output by using the output format defined for the variable.

3. Create variables by executing the FDP$CREATE_VARIABLE procedure. Data is passed to and from the program using variables.

2. Forms can be created with CYBIL, but not with COBOL, FORTRAN, Pascal, or SCL.

4. Create groups of variables that occur more than once by executing the FDP$CREATE_TABLE procedure. You can store more variables than can be shown on the screen at one time. The table can be created before all the variables have been created. All the variables in the table must be created before a FDP$END_FORM procedure is executed. Execute a FDP$CREATE_OBJECT procedure for each table occurrence visible on the form. If you want to specify an initial value of an occurrence that does not appear initially on the form, you can accept the default value of the first occurrence in the table or execute the FDP$CREATE_STORED_OBJECT procedure.

5. Change the record definition (containing attributes) that is used to transfer variables between the program and Screen Formatting by executing the FDP$CHANGE_FORM_RECORD procedure.

   The attributes affected can be:

   - The SCU deck name. If you don't specify a name, the form name is used.

   - The record definition name. In COBOL, the record definition is a COBOL 01-level data name; in CYBIL, Pascal, or SCL, it is a record type name; in FORTRAN, it is an EQUIVALENCE statement.

6. End the form definition by executing the FDP$END_FORM procedure.

7. Write the form definition to a file by executing the FDP$WRITE_FORM_DEFINITION procedure. You can now save the form on an object library.

   The file attributes must have values to be processed by the CREATE_OBJECT_LIBRARY utility. The file content attribute must be set to SCREEN (AMC$SCREEN) and the file structure attribute must be set to FORM (AMC$FORM). Use the CREATE_OBJECT_LIBRARY utility subcommands ADD_MODULE, COMBINE_MODULE, REPLACE_MODULE, and DELETE_MODULE to update the library.

8. Write the record definition to permanent storage by executing the FDP$WRITE_RECORD_DEFINITION procedure.

9. You can now interact with the form by issuing Screen Formatting requests. The first request must be to open the form. Then you can call any procedures to that manipulate the form.

10. When you have finished interacting with the form, close it by executing the FDP$CLOSE_FORM procedure.

Changing a Form

The general steps for changing an existing form definition are as follows:

1. If the form exists on an object library, open the form with the FDP$OPEN_FORM procedure, copy the form with the FDP$COPY_FORM procedure, and call the FDP$EDIT_FORM procedure. If the form was created with the FDP$CREATE_FORM procedure, call the FDP$EDIT_FORM procedure.

2. Get the desired attributes of the form by using the FDP$GET_FORM_ATTRIBUTES procedure. This request can also tell you the number of objects in the form image.

3. Allocate an array for the object definitions and execute the FDP$GET_FORM_OBJECTS procedure to obtain the objects. You can also get names of tables and variables for a form by using the FDP$GET_FORM_NAMES procedure. You can change the attributes associated with tables and variables.

4. Change the form attributes by using the FDP$CHANGE_FORM procedure. You can add, replace, or delete attributes associated with the form.

5. Get the variable attributes by executing the FDP$GET_VARIABLE_ATTRIBUTES procedure. Change the variable attributes by executing the FDP$CHANGE_VARIABLE procedure. You can add, replace, or delete attributes associated with the variable. The variable name can be changed.

6. Get the table attributes with the FDP$GET_TABLE_ATTRIBUTES procedure. Change the table attributes with the FDP$CHANGE_TABLE procedure. You can add, replace, or delete attributes associated with the table. The table name can be changed.

7. Get the object attributes on the form image with the FDP$GET_OBJECT_ATTRIBUTES procedure. Change the attributes of an object on the form image with the FDP$CHANGE_OBJECT procedure. You can add, replace, or delete attributes associated with the object. The position of the object can be changed.

8. Delete an object at a particular form position with the FDP$DELETE_OBJECT procedure. This does not update any related tables or variables.

   Delete a table by using the FDP$DELETE_TABLE procedure. Variables and objects associated with the table are not deleted.

   Delete a variable by using the FDP$DELETE_VARIABLE procedure. This does not update any related table or objects.

9. Get the definitions for the form record by using the FDP$GET_RECORD_ATTRIBUTES procedure. Change the definitions for the form record by using the FDP$CHANGE_RECORD_ATTRIBUTES procedure.

10. Use the FDP$END_FORM procedure to check the form for consistency and to end the form definition. To make further changes to the form definition, you must issue a FDP$EDIT_FORM procedure.

11. Save the changed form by using the FDP$WRITE_FORM_DEFINITION procedure.

    Save the changed record definition with the FDP$WRITE RECORD_DEFINITION procedure.

12. Close the copied form with the FDP$CLOSE_FORM procedure.

    Close the original form with the FDP$CLOSE_FORM procedure.

Designing a Form Interactively

You can write a program that interacts with the user to create, display, or change a form. The program might offer the choice of background color for the form, or might even allow the user to select the language of its text (French, German, etc.). From the programmer's point of view, this method is much more complicated than the dynamic method discussed earlier, but has the advantage of allowing the user to modify the form as desired.

Creating a Form

To have a form created interactively, use two forms:

- The design form interactively helps the user create a form.
- The target form is the one the user wants to create.

Each form has different properties. The design form has events such as SAVE, MARK, and DEFINE that help the user and your program design a form. SAVE collects all the information for a target form and stores it on an object library for future use. MARK displays text with distinctive display attributes so that the user can recognize what text will be affected by a command such as copy, move, or define. DEFINE allows the user to specify attributes for the marked text.

The target form, of course, has the events needed by the application user.

The text on the two forms can also have different display attributes. This allows you to protect text on the target form while allowing the user to type the desired text (and change it) on the design form. The position of the text can be the same on both forms. Use this method of protecting text on the target form and not on the design form for constant text objects.

You use the opposite method of protecting text on the design form. Also for not protecting the same text on the target form for variable text objects that have special attributes, such as inverse video. You must protect text on the design form that requires special attributes. The protection prevents any changes to the special attributes for the object. However, you must allow the user of the target form to enter data into the same object in that form.

Some terminals cannot protect screen text. If not, Screen Formatting restores any modified text that is supposed to be protected after the user transmits the data. When your form design application must handle these terminals, an additional problem appears. The user can modify any text on the design form. However, some of the text can be logically protected by Screen Formatting. When the user transmits the data, some of the changed text is restored to protected value.

One way to alleviate the problem of users changing protected text is to provide a display attribute which allows the user to recognize protected areas. The form attribute FDC$DESIGN_DISPLAY_ATTRIBUTE does this. When an object on a design form does not have an attribute, it takes on this attribute by default.

The general steps for designing a form interactively are:

- Create a design form. Define events that allow the user to specify attributes for text. Have the events cause your program to display other forms on which the user specifies the attributes.
- Create a target form. A profile of the user's application can help specify many of the values for the target form and reduce the amount of user input.
- Have the user create objects on the design form and on the target form, as needed. Users type in text on the design form as it should appear on the target form.
- Give the user a menu of events that perform special events on the objects being created. For example, the DEFINE event should create objects on both the design and target forms with individual display attributes. In the case of creating a variable on the target form to be used for input, create a constant text object on the design form and a variable text object on the target form.
- When the user wants to save the target form, create constant text objects for the target form from the unprotected text on the design form.

The following steps describe this process in more detail.

1. Create a design form by using the FDP$CREATE_DESIGN_FORM procedure. Specify form attributes just as on the FDP$CREATE_FORM procedure.

   The FDP$CREATE_DESIGN_FORM procedure, however, does not need a FDP$END_FORM procedure to signal the completion of its definition. Before displaying the design form on the terminal screen you need to issue FDP$OPEN_FORM and FDP$ADD_FORM procedures.

   The FDP$CREATE_DESIGN_FORM procedure creates a table and a variable that allow you to access all characters on the design form. The name of the variable is specified by using the form attribute FDC$DESIGN_VARIABLE_NAME. If you do not specify this form attribute, the variable name is given by FDC$SYSTEM_DESIGN_VARIABLE_NAME.

   You can use the FDP$GET_STRING_VARIABLE and FDP$REPLACE_STRING_VARIABLE procedures to access characters on the design form. The length of the variable is the same as the width of the design form. The table has as many occurrences as the height of the design form.

2. Create a target form by using the FDP$CREATE_FORM procedure.

3. Open the design form by using the FDP$OPEN_FORM procedure.

4. Schedule the design form for display by using the FDP$ADD_FORM procedure. The next FDP$READ_FORMS procedure displays the design form.

5. Place initial text on the design form. The initial text might come from information the user specified earlier in the application profile. Any text the user can modify by typing over is placed on the design form with the FDP$REPLACE_STRING_VARIABLE procedure. Text which must be protected is placed on the design form with the FDP$CREATE_OBJECT procedure.

   You cannot create any variable text objects on the design form. Any text created by the FDP$CREATE_OBJECT procedure is also stored in the design form and can be retrieved by using the FDP$GET_STRING_VARIABLE procedure.

6. Update the screen and read the design form by using the FDP$READ_FORMS procedure.

7. Get the events the user executed by using the FDP$GET_NEXT_EVENT procedure.

   a. If the user executes a SAVE form event:

      1) Collect the unprotected text on the design form by using the FDP$CREATE_CONSTANT_TEXT procedure. This creates constant objects with no attributes for the target form. Any protected text on the design form is ignored.

      2) Use the FDP$END_FORM procedure to check the form for consistency and to end the form definition. This procedure returns the errors in a sequence. Screen Formatting organizes the data for the form for efficient processing of form interaction requests. To make further changes to the form definition, you must issue a FDP$EDIT_FORM procedure.

      3) Write the form to permanent storage by using the FDP$WRITE_FORM_DEFINITION procedure. Update the object library containing the user's forms.

b. If the user executes a MARK text event:

   1) Save the position of the event. This is the beginning of the text.

   2) Issue a FDP$CREATE_MARK procedure to show the user the beginning of the marked text.

   3) Read the design form by using the FDP$READ_FORMS procedure. The screen is updated and the user can see the mark.

   4) Get the next event the user executes by using the FDP$GET_NEXT_EVENT procedure. In this case, assume the user executes another mark event.

   5) Save the position of the event. This is the end of the text.

   6) Use a FDP$CREATE_MARK procedure to show the user the full area of text selected.

   7) Update the form and get the users next input event by using the FDP$READ_FORMS procedure.

c. If the user executes a DEFINE variable event, do the following:

   1) Conduct a dialogue with the user to obtain additional information about the variable. For instance, the user may want to specify the variable name, the program data type, and the terminal input and output actions. The marked text on the design form gives the position, length and initial value of the variable. Create the variable for the target form by using the FDP$CREATE_VARIABLE procedure.

   2) Protect the text representing the variable on the design form by creating a constant text object with the FDP$CREATE_OBJECT procedure. Also create a variable text object on the target form with the FDP$CREATE_OBJECT procedure.

d. If the user executes a DELETE mark event, clear any program pointers to marked text, issue the FDP$DELETE_MARK procedure, and read the design form by using the FDP$READ_FORMS procedure.

e. If the user executes a move event, do the following. Assume that the user had previously marked the area to be moved and moved the cursor to the desired destination when executing the move event.

   1) Move the objects on the design form by using the FDP$MOVE_AREA procedure. On the design form, both constant text objects (protected text) and unprotected text will then be moved. Move the objects on the target form by using the FDP$MOVE_AREA procedure.

   2) Update the screen with the FDP$READ_FORMS procedure.

f. If the user executes a copy event, do the following. Assume that the user had previously marked the area to be copied and then moved the cursor to the desired destination when executing the copy event.

   1) Copy the objects on the design form by using the FDP$COPY_AREA procedure. On the design form, both constant text objects and unprotected text will then be copied. Copy the objects on the target form by using the FDP$COPY_AREA procedure.

   2) Update the screen with the FDP$READ_FORMS procedure.

Changing a Form

The steps for changing a form are as follows:

1. Open the form by using the FDP$OPEN_FORM procedure.

2. Copy the form by using the FDP$COPY_FORM procedure. The output of the FDP$COPY_FORM procedure is the target form.

3. Indicate that you wish to change the target form by using the FDP$EDIT_FORM procedure.

4. Create the design form by using the FDP$CREATE_DESIGN_FORM procedure.

5. Create the initial data on the design form. The FDP$CREATE_DESIGN_TEXT procedure creates constant text objects (protected text), line drawings (protected), and unprotected text on the design form from the target form. Constant text objects with attributes on the target form will be represented as constant text objects on the design form. Variables on the target form will be represented as constant text objects using their initial value on the design form. If the variable has no display attributes, the display attributes specified by the form attribute FDC$DESIGN_DISPLAY_ATTRIBUTE will be used. The FDC$DESIGN_DISPLAY_ATTRIBUTE helps the form designer to recognize variables.

   Constant text objects without any attributes will be represented as unprotected text on the design form. Objects in the target form representing unprotected text on the design form are deleted from the target form. When the user saves the form, the constant text objects for the target form will be created using the unprotected text from the design form.

6. Schedule the design form for display by using the FDP$ADD_FORM procedure.

7. Read the design form by using the FDP$READ_FORMS procedure. The user may freely modify unprotected text (such as form titles, variable labels, and directions). The user executes events to change protected text.

8. Get the events the user executed by using the FDP$GET_NEXT_EVENT procedure. Many of the events described in the section on creating a form also occur when changing a form. The following steps highlight events that occur when changing a form.

   a. If the user executes a DELETE event:

      1) Delete the object from the design form by using the FDP$DELETE_OBJECT procedure. Any text on the design form associated with the object is set to spaces. Delete the object from the target form as well.

         If the object was a variable text object, it should also be deleted from the target form with the FDP$DELETE_VARIABLE procedure. In addition, if it was defined in a table it should be deleted from the table with the FDP$CHANGE_TABLE procedure.

      2) Update the screen and get the user's next input by using the FDP$READ_FORMS procedure.

   b. If the user executes a change event:

      1) Get the current attributes of the object by using the appropriate FDP$GET_OBJECT_ATTRIBUTES, FDP$GET_VARIABLE_ATTRIBUTES, and FDP$GET_TABLE_ATTRIBUTES procedures.

      2) Conduct a dialogue with the user to learn the desired change. Show the user the current attributes. Allow the user to change only the attributes that the user desires.

      3) Change the object on the design form by using the FDP$CHANGE_OBJECT procedure. Any text on the design form is also changed. An FDP$GET_STRING_VARIBLE procedure would see the changed text. Change the object on the target form by using the appropriate FDP$CHANGE_OBJECT, FDP$CHANGE_TABLE, and FDP$CHANGE_VARIABLE procedures.

Displaying a Form

If the user wants to view a form to evaluate changes, your design form program will have to display previously saved forms. Define one consistent event for the user to execute in order to end the viewing of any form handled by your program. This means you have to change the events originally defined for the form. To do this, program the following steps.

1. Open the desired form by using the FDP$OPEN_FORM procedure.
2. Copy the form to storage that can be modified by using the FDP$COPY_FORM procedure.
3. Begin editing of the copied form by using the FDP$EDIT_FORM procedure.
4. Change the events associated with the copied form by using the FDP$CHANGE_FORM procedure. You delete all previous events by using the form attribute FDC$DELETE_ALL_EVENTS. Define one event that the user executes to terminate viewing of the form.
5. End the form changes for the copied form by using the FDP$END_FORM procedure.
6. Open the copied form by using the FDP$OPEN_FORM procedure.
7. Schedule the copied form for display by using the FDP$ADD_FORM procedure.
8. Display the form by using the FDP$READ_FORMS procedure.
9. Learn when the user wants to finish viewing the form by using the FDP$GET_NEXT_EVENT procedure. When the user executes the display termination event, close the opened form and the copied form by using the FDP$CLOSE_FORM procedure. Otherwise, continue displaying the form.

Rectangle Form Program

The following example shows a program that creates the form and form definition record for Rectangle Form (used in the CYBIL program in chapter 6, Using CYBIL to Manage Forms). The program (including compiling information) is also in the Examples online manual. Look for it under the name CYBIL_CREATE_RECTANGLE_FORM in the Screen Formatting examples.

```
?? RIGHT := 110 ??
MODULE create_rectangle_form;
*copyc amp$get_segment_pointer
*copyc fsp$close_file
*copyc fsp$open_file
*copyc amp$set_segment_eoi
*copyc fdp$close_form
*copyc fdp$create_form
*copyc fdp$create_object
*copyc fdp$create_table
*copyc fdp$create_variable
*copyc fdp$end_form
*copyc fdp$write_form_definition
*copyc fdp$write_record_definition
*copyc pmp$abort

  PROGRAM create_rectangle_form
    (VAR status: ost$status);

    VAR
      area_variable_name: [READ] ost$name := 'AREA',
       display_name: [READ] ost$name := 'ERROR',
      display_attribute: [READ] fdt$display_attribute_set :=
            $fdt$display_attribute_set [fdc$inverse_video],
      file_cycle_attributes: [STATIC] array [1 .. 1] of
            fst$file_cycle_attribute :=
            [[fsc$file_contents_and_processor,
            fsc$screen_form, osc$null_name]],
      form_attributes: array [1 .. 6] of fdt$form_attribute,
      form_fid: amt$file_identifier,
      form_identifier: fdt$form_identifier,
       form_file: [STATIC] string (18) := '$LOCAL.FORM_BINARY',
      form_name: [READ] ost$name := 'CYBIL_RECTANGLE_FORM',
      local_status: ost$status,
      number_errors: fdt$number_errors,
      message_variable_name: [READ] ost$name := 'MESSAGE',
```

```
      object_attributes: array [1 .. 2] of fdt$object_attribute,
      object_definition: fdt$object_definition,
      p_errors: ^SEQ ( * ),
       record_file: [STATIC] string (18) := '$LOCAL.FORM_RECORD',
      record_fid: amt$file_identifier,
      segment_pointer: amt$segment_pointer,
      side_table_name: [READ] ost$name := 'SIDE_TABLE',
      side_variable_name: [READ] ost$name := 'SIDE',
      table_attributes: array [1 .. 2] of fdt$table_attribute,
      text: string (80),
      variable_attributes: array [1 .. 2] of
            fdt$variable_attribute;

{   Define normal events.

    form_attributes [1].key := fdc$add_event;
    form_attributes [1].event_action :=
          fdc$return_program_normal;
    form_attributes [1].event_name := 'COMPUTE';
    form_attributes [1].event_label := 'Comput';
    form_attributes [1].event_trigger := fdc$next;

{ Define abnormal events.

    form_attributes [2].key := fdc$add_event;
    form_attributes [2].event_action :=
          fdc$return_program_abnormal;
    form_attributes [2].event_name := 'QUIT';
    form_attributes [2].event_label := 'Quit';
    form_attributes [2].event_trigger := fdc$stop;

    form_attributes [3].key := fdc$add_event;
    form_attributes [3].event_action :=
          fdc$return_program_abnormal;
    form_attributes [3].event_name := 'BACK';
    form_attributes [3].event_label := 'Back';
    form_attributes [3].event_trigger := fdc$back;

{ Define form name.

    form_attributes [4].key := fdc$form_name;
    form_attributes [4].form_name := form_name;
```

```
{ Define event form.

    form_attributes [5].key := fdc$event_form;
    form_attributes [5].event_form_definition.key :=
          fdc$system_default_event_form;

{ Define program display attribute.

    form_attributes [6].key := fdc$add_display_definition;
    form_attributes [6].display_attribute := display_attribute;
    form_attributes [6].display_name := display_name;

    fdp$create_form (form_identifier, form_attributes, status);
    IF NOT status.normal THEN
      pmp$abort (status);
    IFEND;

{   Create variable for side.

    variable_attributes [1].key := fdc$program_data_type;
    variable_attributes [1].program_data_type :=
          fdc$program_integer_type;

    variable_attributes [2].key := fdc$unused_variable_entry;
    fdp$create_variable (form_identifier, side_variable_name,
          variable_attributes, status);
    IF NOT status.normal THEN
      pmp$abort (status);
    IFEND;

{ Create variable for area.

    variable_attributes [2].key := fdc$io_mode;
    variable_attributes [2].io_mode := fdc$terminal_output;
    fdp$create_variable (form_identifier, area_variable_name,
          variable_attributes, status);
    IF NOT status.normal THEN
      pmp$abort (status);
    IFEND;
```

```
{ Create variable for message.

    variable_attributes [1].key := fdc$unused_variable_entry;
    fdp$create_variable (form_identifier, message_variable_name,
          variable_attributes, status);
    IF NOT status.normal THEN
      pmp$abort (status);
    IFEND;

{ Create table containing rectangle sides.

    table_attributes [1].key := fdc$stored_occurrence;
    table_attributes [1].stored_occurrence := 2;

    table_attributes [2].key := fdc$add_table_variable;
    table_attributes [2].variable_name := side_variable_name;
    fdp$create_table (form_identifier, side_table_name,
          table_attributes, status);
    IF NOT status.normal THEN
      pmp$abort (status);
    IFEND;

{ Create constant text objects.

    object_attributes [1].key := fdc$unused_object_entry;
    object_attributes [2].key := fdc$unused_object_entry;
    text := 'Compute Area of Rectangle:';
    object_definition.key := fdc$constant_text;
    object_definition.p_constant_text := ^text (1, 26);
    object_definition.constant_text_width := 26;
    fdp$create_object (form_identifier, 20, 5, object_definition,
          object_attributes, status);
    IF NOT status.normal THEN
      pmp$abort (status);
    IFEND;
```

```
    text := 'Type height:';
    object_definition.p_constant_text := ^text (1, 12);
    object_definition.constant_text_width := 12;
    fdp$create_object (form_identifier, 52, 9, object_definition,
          object_attributes, status);
    IF NOT status.normal THEN
      pmp$abort (status);
    IFEND;

    text := 'Type width:';
    object_definition.p_constant_text := ^text (1, 11);
    object_definition.constant_text_width := 11;
    fdp$create_object (form_identifier, 20, 11,
          object_definition, object_attributes, status);
    IF NOT status.normal THEN
      pmp$abort (status);
    IFEND;

    text := 'Area is:';
    object_definition.p_constant_text := ^text (1, 8);
    object_definition.constant_text_width := 8;
    fdp$create_object (form_identifier, 20, 9, object_definition,
          object_attributes, status);
    IF NOT status.normal THEN
      pmp$abort (status);
    IFEND;

{ Create box.

    object_definition.key := fdc$box;
    object_definition.box_width := 36;
    object_definition.box_height := 4;
    object_attributes [1].key := fdc$unused_object_entry;
    object_attributes [2].key := fdc$unused_object_entry;
    fdp$create_object (form_identifier, 15, 7, object_definition,
          object_attributes, status);
    IF NOT status.normal THEN
      pmp$abort (status);
    IFEND;
```

```
{ Create variable text for height (side [1]).

    object_definition.key := fdc$variable_text;
    object_definition.variable_text_width := 10;
    text (1, 10) := ' ';
    object_definition.p_variable_text := ^text (1, 10);
    object_attributes [1].key := fdc$object_name;
    object_attributes [1].object_name := side_variable_name;
    object_attributes [1].occurrence := 1;
    object_attributes [2].key := fdc$object_display;
    object_attributes [2].display_attribute :=
          $fdt$display_attribute_set [fdc$underline];
    fdp$create_object (form_identifier, 65, 9, object_definition,
          object_attributes, status);
    IF NOT status.normal THEN
      pmp$abort (status);
    IFEND;

{ Create variable text for width (side [2]).

    object_attributes [1].occurrence := 2;
    fdp$create_object (form_identifier, 32, 11,
          object_definition, object_attributes, status);
    IF NOT status.normal THEN
      pmp$abort (status);
    IFEND;

{ Create variable text for area.

    object_attributes [1].object_name := area_variable_name;
    object_attributes [1].occurrence := 1;
    object_attributes [2].key := fdc$unused_object_entry;
    fdp$create_object (form_identifier, 29, 9, object_definition,
          object_attributes, status);
    IF NOT status.normal THEN
      pmp$abort (status);
    IFEND;
```

```
{ Create variable text for message.

    object_attributes [1].object_name := message_variable_name;
    text (1, 40) := ' ';
    object_definition.variable_text_width := 40;
    object_definition.p_variable_text := ^text (1, 40);
    fdp$create_object (form_identifier, 20, 15,
          object_definition, object_attributes, status);
    IF NOT status.normal THEN
      pmp$abort (status);
    IFEND;

    fdp$end_form (form_identifier, NIL, number_errors, p_errors,
          status);
    IF NOT status.normal THEN
      pmp$abort (status);
    IFEND;
    IF number_errors <> 0 THEN
      pmp$abort (status);
    IFEND;

{ Write binary form definition for object code library.

    fsp$open_file (form_file, amc$segment, NIL, NIL,
          ^file_cycle_attributes, NIL, NIL, form_fid, status);
    IF NOT status.normal THEN
      pmp$abort (status);
    IFEND;

    amp$get_segment_pointer (form_fid, amc$sequence_pointer,
          segment_pointer, status);
    IF NOT status.normal THEN
      pmp$abort (status);
    IFEND;
```

```
     RESET segment_pointer.sequence_pointer;
    fdp$write_form_definition (form_identifier,
          segment_pointer.sequence_pointer, status);
    IF NOT status.normal THEN
      pmp$abort (status);
    IFEND;

    amp$set_segment_eoi (form_fid, segment_pointer, status);
    fsp$close_file (form_fid, local_status);
    IF NOT status.normal THEN
      pmp$abort (status);
    IFEND;

{ Write CYBIL record definition for source code library.

    fsp$open_file (record_file, amc$record, NIL, NIL,  NIL, NIL,
         NIL, record_fid, status);
    IF NOT status.normal THEN
      pmp$abort (status);
    IFEND;

    fdp$write_record_definition (form_identifier, record_fid,
          fdc$cybil_processor, status);
    IF NOT status.normal THEN
      pmp$abort (status);
    IFEND;

     fsp$close_file (record_fid, status);
    fdp$close_form (form_identifier, status);

  PROCEND create_rectangle_form;

MODEND create_rectangle_form;
```

Creating Form Definition Records for Existing Forms

Form definition records define the variables created on forms. The definitions are written in the language of the program managing the forms. The application programmer must include these definitions in the program. Usually, these definitions are maintained on a Source Code Utility (SCU) library and are copied into the program when it is expanded. When creating forms using a CYBIL program, you include a call to the FDP$WRITE_RECORD_DEFINITION procedure. You can also use this procedure to create form definition records for existing forms. For an existing form, open the form and then call FDP$WRITE_RECORD_DEFINITION to create the form definition.

If the form was created using the Screen Design Facility (SDF), you can still create a form definition record using the FDP$WRITE_RECORD_DEFINITION procedure in a CYBIL program. At this time SDF cannot create record definitions for use with Pascal application programs. The only method available, besides manually creating them, is to use a CYBIL program.

In the online Examples manual, there is a CYBIL program that will create form definition records for COBOL, FORTRAN, Pascal, SCL or CYBIL. The procedure that executes the program has parameters for the name of the file to contain the record, the name of the form (the form must be accessible through your command list), and the name of the language you want the definition written in. To use the program, find the example named CREATE_RECORD in the Screen Formatting examples. The example creates one record each time you execute it.

Attributes for a Form

When defining a form using Screen Formatting, you must define its attributes. These attributes can be categorized as:

- Basic form attributes

  Attributes that describe the appearance of the form and its events.

- Variable attributes

  Attributes that describe variables on a form; for example, the data types of the variables.

- Table attributes

  Attributes that describe tables containing variables, for example, the number of occurrences of the variables in the table.

- Object attributes

  Attributes for objects that appear on the form; for example, the name and position of objects. Objects can be either text or graphics.

- Record attributes

  Attributes for form definition records, including the name.

You can add new attributes, replace attributes, and delete attributes, as well as accept default attributes. You can also retrieve the current attributes.

These attributes are contained in an array of records, each attribute stored as a value in a separate record. You must initialize this value to the desired attribute in order to create, change, or get a specific attribute.

Basic Form Attributes

These attributes define the appearance of the form, its events, and several other form characteristics. They are in two groups, those for creating or changing the form and those for returning the current values of the form.

Creating and Changing Forms

The following attributes are for creating and changing forms. As stated earlier, each attribute is specified as a value in a record in an initialized array. Each record is of type FDT$FORM_ATTRIBUTE, which is listed in appendix F.

Once established, this array is named on the FORM_ATTRIBUTES parameter in the call to any of the following CYBIL procedures, which are described later in this chapter:

FDP$CHANGE_FORM
FDP$CREATE_FORM
FDP$CREATE_DESIGN_FORM
FDP$CREATE_EVENT_FORM

The following are the attribute records, their descriptions, and the values permitted for each. The attribute record names are in italics.

add_event

Specifies that an event is added to the list of events for a form. This record contains the following fields: event_name, event_label, event_trigger, and event_action.

event_name

The name of the event that the application programs use (type OST$NAME). It must be unique and follow the form processor language conventions. Examples are copy, delete, and add. The event name is also the variable name on an event form associated with this form.

event_label

The event-label displayed at the bottom of the screen (type OST$NAME). Only the first 6 characters of the label are used. For the standard events defined by Control Data, use the following labels.

Standard Event	Label
Move backward	Bkw
Move to first	First
Move forward	Fwd
Move to last	Last
Back to previous context	Back
Request help	Help
Undo last event	Undo
Redo last event	Redo
Quit save	Quit
Alternate exit	Exit

event_trigger

An ordinal specifying the terminal event (type FDT$EVENT_TRIGGER). Terminal events correspond to keys that can be specified in the terminal definition. Screen Formatting assigns a key when a key does not exist in the terminal definition. If a terminal definition key does not have a label, the key is assumed not to exist. But even if a key cannot be assigned, the user is still permitted to interact with the form. The following rules are used to assign keys:

- First, the event triggers are assigned to their corresponding keys using the priority given after these rules.

- If an event trigger cannot be assigned, the following steps are executed:

  1. Assigns standard event triggers (FDC$NEXT, FDC$SHIFT_NEXT, .. FDC$SHIFT_DATA) to unused terminal function keys (FDC$FUNCTION_1, FDC$SHIFT_FUNCTION_1, .. FDC$SHIFT_FUNCTION_16).

2. Assigns application event triggers to unused terminal function keys in ascending order of function number. This means that FDC$FUNCTION_1, FDC$SHIFT_FUNCTION_1 is assigned before FDC$FUNCTION_2, FDC$SHIFT_FUNCTION_2. (Triggers are assigned to the same key, whether shifted or unshifted, if possible.)

3. Assigns non-shifted event triggers to non-shifted unused terminal function keys. Screen Formatting tries to assign shifted event triggers to shifted unused terminal function keys.

4. Assigns keys while opening the form. By using the FDP$GET_FORM_ATTRIBUTES request with the key FDC$GET_NEXT_EVENT, you can learn the keys (event trigger) that Screen Formatting assigned.

The priority in which terminal definition keys are assigned is as follows:

FDC$NEXT
FDC$SHIFT_NEXT
FDC$HELP
FDC$SHIFT_HELP
FDC$STOP
FDC$SHIFT_STOP
FDC$BACK
FDC$SHIFT_BACK
FDC$UP
FDC$SHIFT_UP
FDC$DOWN
FDC$SHIFT_DOWN
FDC$FORWARD
FDC$SHIFT_FORWARD
FDC$BACKWARD
FDC$SHIFT_BACKWARD
FDC$UNDO
FDC$REDO
FDC$EDIT
FDC$SHIFT_EDIT
FDC$DATA
FDC$SHIFT_DATA
FDC$FUNCTION_1
FDC$SHIFT_FUNCTION_1
FDC$FUNCTION_2
FDC$SHIFT_FUNCTION_2
FDC$FUNCTION_3

FDC$SHIFT_FUNCTION_3
FDC$FUNCTION_4
FDC$SHIFT_FUNCTION_4
FDC$FUNCTION_5
FDC$SHIFT_FUNCTION_5
FDC$FUNCTION_6
FDC$SHIFT_FUNCTION_6
FDC$FUNCTION_7
FDC$SHIFT_FUNCTION_7
FDC$FUNCTION_8
FDC$SHIFT_FUNCTION_8
FDC$FUNCTION_9
FDC$SHIFT_FUNCTION_9
FDC$FUNCTION_10
FDC$SHIFT_FUNCTION_10
FDC$FUNCTION_11
FDC$SHIFT_FUNCTION_11
FDC$FUNCTION_12
FDC$SHIFT_FUNCTION_12
FDC$FUNCTION_13
FDC$SHIFT_FUNCTION_13
FDC$FUNCTION_14
FDC$SHIFT_FUNCTION_14
FDC$FUNCTION_15
FDC$FUNCTION_16
FDC$SHIFT_FUNCTION_16
FDC$PICK
FDC$INSERT_LINE
FDC$DELETE_LINE
FDC$HOME_CURSOR
FDC$CLEAR_SCREEN
FDC$TIME_OUT
FDC$VARIABLE_TRIGGER

Screen Formatting supports standard events. A standard event is one that has a label defined by Control Data and performs an event defined by Control Data.

The system assigns the standard events as follows:

1. The application must use the standard event if it exists for the event being defined.

2. If a terminal has a dedicated key that performs the standard event, the standard event is assigned to that key.

3. If a terminal does not have a dedicated key that performs the standard event, the standard event is assigned either to a key such as a programmable function key or to a sequence of keys defined by the terminal definition.

The following table lists the trigger for each standard event.

Standard Event	Screen Formatting Trigger
Move backward	FDC$BACKWARD
Move to first	FDC$SHIFT_BACKWARD/ FDC$FIRST
Move forward	FDC$FORWARD
Move to last	FDC$SHIFT_FORWARD/ FDC$LAST
Back to previous context	FDC$BACK
Request help	FDC$HELP
Undo last event	FDC$UNDO
Redo last event	FDC$REDO
Quit save	FDC$STOP/ FDC$QUIT
Alternate exit	FDC$SHIFT_STOP/ FDC$EXIT

If you specify one of the alternate forms (FDC$FIRST, FDC$LAST, FDC$QUIT, FDC$EXIT) for the trigger, the primary form is stored (FDC$SHIFT_BACKWARD, FDC$SHIFT_FORWARD, FDC$STOP, FDC$SHIFT_STOP). That means any request that returns a trigger returns the primary form.

event_action

Specifies a variant record of type FDT$EVENT_ACTION containing one of the following values. In the descriptions that follow, "current" refers to the table or variable on which the cursor is positioned. With the exception of FDC$RETURN_PROGRAM_NORMAL and FDC$RETURN_PROGRAM_ABNORMAL[3], the event is not returned to the program.

FDC$RETURN_PROGRAM_NORMAL

Returns to the program, indicating that the event is normal.

3. For a definition of normal versus abnormal events, refer to Processing Events and Data in chapter 4.

FDC$RETURN_PROGRAM_ABNORMAL

Returns to the program, indicating that the event is abnormal.

FDC$PAGE_TABLE_FORWARD

The current table pages forward to its next group of occurrences.[4]

FDC$PAGE_TABLE_BACKWARD

The current table pages backward to its previous group of occurrences.

FDC$SCROLL_TABLE_FORWARD

The current table scrolls forward.

FDC$SCROLL_TABLE_BACKWARD

The current table scrolls backward.

FDC$DISPLAY_HELP

The help information is displayed for either the form or the current variable.

FDC$ERASE_HELP

The help information displayed on the screen is erased.

FDC$EXECUTE_COMMAND

Unused.

FDC$IGNORE_EVENT

This event is ignored. It is not returned to the application program.

FDC$TAB_TO_NEXT_FORM_FIELD

The cursor moves to the next input variable on the form. If the cursor is on the form's last variable, it moves to the first input variable. This differs from tabbing to the next unprotected field, which works over the entire screen rather than within a form.

4. Occurrences are explained earlier in this chapter under Tables.

FDC$TAB_TO_PREVIOUS_FORM_FIELD

The cursor moves to the previous variable. If the cursor is on the form's first variable, it moves to the last input variable. This differs from tabbing to the previous unprotected field, which works over the entire screen rather than within a form.

FDC$SCROLL_VARIABLE_FORWARD

The current variable scrolls forward.

FDC$SCROLL_VARIABLE_BACKWARD

The current variable scrolls backward.

FDC$PAGE_VARIABLE_FORWARD

The current variable pages forward to its next group of characters.

FDC$PAGE_VARIABLE_BACKWARD

The current variable pages backward to its previous group of characters.

FDC$PAGE_VARIABLE_FIRST

Pages to the first portion of the variable.

FDC$PAGE_VARIABLE_LAST

Pages to the last portion of the variable.

FDC$PAGE_TABLE_FIRST

Pages to the first group of occurrences within a table.

FDC$PAGE_TABLE_LAST

Pages to the last group of occurrences within a table.

add_form_comment

Currently unused.

add_display_definition

Specifies the set of attributes called the display definition, which allows a program to change the display characteristics of a form object. This record contains two fields:

display_attribute

A set of display attributes (type FDT$DISPLAY_ATTRIBUTE_SET). Possible values are:

FDC$INVERSE_VIDEO
FDC$LOW_INTENSITY
FDC$HIGH_INTENSITY
FDC$BLINK
FDC$UNDERLINE
FDC$PROTECT
FDC$HIDDEN
FDC$BLACK_FOREGROUND
FDC$BLUE_FOREGROUND
FDC$GREEN_FOREGROUND
FDC$MAGENTA_FOREGROUND
FDC$RED_FOREGROUND
FDC$CYAN_FOREGROUND
FDC$YELLOW_FOREGROUND
FDC$WHITE_FOREGROUND
FDC$BLACK_BACKGROUND
FDC$BLUE_BACKGROUND
FDC$GREEN_BACKGROUND
FDC$MAGENTA_BACKGROUND
FDC$RED_BACKGROUND
FDC$CYAN_BACKGROUND
FDC$YELLOW_BACKGROUND
FDC$WHITE_BACKGROUND
FDC$FINE_LINE
FDC$MEDIUM_LINE
FDC$BOLD_LINE
FDC$ITALIC_DISPLAY_ATTRIBUTE
FDC$TITLE_DISPLAY_ATTRIBUTE
FDC$INPUT_DISPLAY_ATTRIBUTE
FDC$ERROR_DISPLAY_ATTRIBUTE
FDC$MESSAGE_DISPLAY_ATTRIBUTE

display_name

The application program name that sets the attribute for an object (type OST$NAME).

delete_all_displays

Deletes all currently defined displays.

delete_all_events

Deletes all events.

delete_event,

delete_display_definition

Deletes the specified event from a list of events, or deletes the specified display definition (type OST$NAME).

delete_form_comments

Currently unused.

design_display_attribute

Specifies the set of display attributes (type FDT$DISPLAY_ATTRIBUTE_SET) to be used with an object on the design form when the object has no attributes assigned. This allows the form designer to recognize the object. The default is FDC$UNDERLINE. For the list of display attributes, refer to add_display_definition earlier in this chapter.

design_variable_name

Specifies the variable name used to access text on a design form (type OST$NAME).

event_form

Specifies the event form definition as a variant record (type FDT$EVENT_FORM_DEFINITION). The form being defined can have an associated event form that shows which terminal events cause program events. This event form can contain program event labels and terminal event labels.

A maximum of 16 terminal function keys can be shown on the event form. Two program event labels can appear for each terminal function key. The upper label is a shifted function key (MARK, for instance, is shifted F1 in the following example).

Here is an example of an event form:

```
   MARK      MOVE             REDO
F1 UNMARK F2 COPY    ...   F8 UNDO
```

F1, F2, ... F8 are terminal function key labels that come from the terminal definition. MARK, UNMARK, MOVE, COPY, ... REDO, UNDO are event labels that come from the form definition.

The KEY field (type FDT$EVENT_FORM_KEY) contains one of the following:

FDC$NO_EVENT_FORM

No event form is generated.

FDC$SYSTEM_DEFAULT_EVENT_FORM

Screen Formatting generates an event form showing application functions.

FDC$USER_EVENT_FORM

The specified event form is used (type OST$NAME).

form_area

Contains a variant record specifying which area of the terminal screen is occupied by the specified form (type FDT$FORM_AREA). Its KEY field (type FDT$FORM_AREA_KEY) contains one of the following (by default, the entire screen is occupied):

FDC$DEFINED_AREA

Indicates the location and size of the rectangle which the form occupies. It contains four fields:

x_position

The x position is determined relative to the top left corner of the screen. The first x position (type FDC$X_POSITION) is one. x increases by one, left to right, for each character. Allowable values are from 1 to 256.

y_position

The y position is determined relative to the top left corner of the screen. The first y position (type FDC$Y_POSITION) is one. y increases by one for each line of the screen from top to bottom. Allowable values are from 1 to 256.

width

The form width (type FDT$WIDTH) is represented as a number greater than or equal to one.

height

The form height (type FDT$HEIGHT) is represented as a number greater than or equal to one.

FDC$SCREEN_AREA

Uses the entire screen. The size of the screen (the number of columns and rows displayed) is determined by the number of lines the form contains and its widest line.

form_display_attribute

Specifies a set of display attributes for the form (type FDT$DISPLAY_ATTRIBUTE_SET). If you don't specify any attributes for an object on the form, the background and foreground attributes associated with this record are used. The default attributes are FDC$BLACK_BACKGROUND and FDC$WHITE_FOREGROUND.

FDC$INVERSE_VIDEO
FDC$BLACK_BACKGROUND
FDC$BLUE_BACKGROUND
FDC$GREEN_BACKGROUND
FDC$MAGENTA_BACKGROUND
FDC$RED_BACKGROUND
FDC$CYAN_BACKGROUND
FDC$YELLOW_BACKGROUND
FDC$WHITE_BACKGROUND
FDC$BLACK_FOREGROUND
FDC$BLUE_FOREGROUND
FDC$GREEN_FOREGROUND
FDC$MAGENTA_FOREGROUND
FDC$RED_FOREGROUND
FDC$CYAN_FOREGROUND
FDC$YELLOW_FOREGROUND
FDC$WHITE_FOREGROUND
FDC$FINE_BORDER
FDC$MEDIUM_BORDER
FDC$BOLD_BORDER

form_help

Specifies a variant record (type FDT$HELP_DEFINITION) for the help information available with the form. This information is provided when the user executes a help event on a form area that contains no object. Its KEY field (type FDT$HELP_KEY) contains one of the following:

FDC$HELP_FORM

The name of an application-defined help form (type OST$NAME).

FDC$HELP_MESSAGE

A pointer to a help message (type ^FDT$HELP_MESSAGE).

FDC$NO_HELP_RESPONSE

Nothing happens when the user executes the help event.

form_language

Currently unused.

form_name

Contains the form name used in the object library (type OST$NAME). You must specify this attribute if you want to save the form on an object library.

form_processor

Specifies the computer language of the program that uses the form (type FDT$FORM_PROCESSOR). You should specify the language before any variable, table, object, or event is created. The default processor is FDC$CYBIL_PROCESSOR. The values are the following:

FDC$ANSI_FORTRAN_PROCESSOR.
FDC$CDC_FORTRAN_PROCESSOR
FDC$COBOL_PROCESSOR
FDC$CYBIL_PROCESSOR
FDC$PASCAL_PROCESSOR
FDC$SCL_PROCESSOR

message_form

Specifies the name of the form designed for error messages (type OST$NAME). This form must be in an object library in the user's command list.

unused_form_entry

Indicates a null filler in the FDT$FORM_ATTRIBUTES array.

validate_variable_values

Specifies validation for initial values of variables on the form. Because this adds a significant amount of processing to the FDP$END_FORM procedure, you may want to include this only when you are debugging the form. Use TRUE to validate variables; otherwise, use FALSE. The default is FALSE.

Getting Basic Form Attributes

The following attribute records return certain values of the form, such as its name or processor. These records are specified in an initialized array. Each record is of type FDT$GET_FORM_ATTRIBUTE, which is listed in Appendix F.

Once established, this array is named on the GET_FORM_ATTRIBUTES parameter in the call to the FDP$GET_FORM_ATTRIBUTES procedure, described later in this chapter.

The following are the attribute records, their descriptions, and the permitted values for each. The attribute record names are in italics.

get_event_form

Returns the event form definition. This record specifies a variant record (type FDT$EVENT_FORM_DEFINITION). Its KEY field (type FDT$EVENT_FORM_KEY) contains one of the following definitions:

FDC$NO_EVENT_FORM

An event form is not generated with the application functions.

FDC$SYSTEM_DEFAULT_EVENT_FORM

An event form is generated with the application functions.

FDC$USER_EVENT_FORM

The event form indicated by this record (type OST$NAME) is used.

get_event_form_identifier

Returns the form identifier of the event form (type FDT$FORM_IDENTIFIER). This identifier can be used in requests to change the value or display attributes of an event label.

get_form_area

Returns the area occupied by the form (type FDT$FORM_AREA). This record specifies a variant record (type FDT$FORM_AREA). Its KEY field (type FDT$FORM_AREA_KEY) contains one of the following:

FDC$DEFINED_AREA

Specifies the location and size of the rectangle which the form occupies. This record returns the following fields:

x_position

The x position is determined relative to the top left corner of the screen. The first x position (type FDC$X_POSITION) is one.

y_position

The y position is determined relative to the top left corner of the screen. The first y position (type FDC$Y_POSITION) is one.

width

The form width (type FDT$WIDTH) is represented as a number greater than or equal to one.

height

The form height (type FDT$HEIGHT) is represented as a number greater than or equal to one.

FDC$SCREEN_AREA

The entire terminal screen is used.

get_form_display_attribute

Returns the set of display attributes used by the form (type FDT$_DISPLAY_ATTRIBUTE_SET). For a list of the display attributes, refer to form_display_attribute earlier in this chapter.

get_form_help

Contains a variant record which returns the help processing available for the form (type FDT$GET_HELP_DEFINITION). Its KEY field (type FDT$GET_HELP_KEY) contains one of:

FDC$GET_HELP_FORM

Returns the name of an application-defined help form (type OST$NAME).

FDC$GET_HELP_MESSAGE

Returns the length of the help message in characters (type FDT$HELP_MESSAGE_LENGTH). Use get_form_help_message to return the help message.

FDC$GET_NO_HELP_RESPONSE

Specifies that Screen Formatting does nothing when the user executes the help event.

get_form_help_message

Contains a pointer (type ^FDT$HELP_MESSAGE) for Screen Formatting to return the help message displayed when the user executes the help event on an area of the form that does not contain an object.

get_form_name

Returns the form name that is used in the object library (type OST$NAME). The default is OSC$NULL_NAME.

get_form_processor

Returns the computer language of the program that uses the form (type FDT$FORM_PROCESSOR). The values are the following:

FDC$ANSI_FORTRAN_PROCESSOR
FDC$CDC_FORTRAN_PROCESSOR
FDC$COBOL_PROCESSOR
FDC$CYBIL_PROCESSOR
FDC$PASCAL_PROCESSOR
FDC$SCL_PROCESSOR

get_next_event

Returns the next event in the list of events for a form. The first occurrence of this record returns the first event, the second returns the second, and so forth. The following events may be returned:

event_action

Refer to the description of the event_action record under the add_event attribute earlier in this chapter (type FDT$EVENT_ACTION).

event_name

Returns the event name (type OST$NAME).

event_command_length

Currently unused.

event_trigger

Refer to the description of the event_trigger record under the add_event attribute earlier in this chapter (type FDT$EVENT_TRIGGER).

get_next_display

Returns the next display definition, which allows a program to change the attributes of a form object. The first occurrence of this record returns the first display attribute, the second returns the second, and so forth. For a description of the fields and of the values returned, refer to add_display_definition earlier in this chapter.

get_number_events

Returns the number of records needed to get the events for the form (type FDT$NUMBER_EVENTS).

get_number_displays

Returns the number of display attributes specified for a form (type FDT$NUMBER_OBJECT_DISPLAYS). For the set of display attributes, refer to the get_next_display record earlier in this section.

get_number_objects

Returns the number of objects on the form (type FDT$NUMBER_OBJECTS).

get_number_tables

Returns the number of tables on the form (type FDT$NUMBER_TABLES).

get_number_variables

Returns the number of form variable definitions created for a particular form (type FDT$NUMBER_VARIABLES). Occurrences created by a table definition are not included.

get_unused_form_entry

Indicates a null filler in the FDT$GET_FORM_ATTRIBUTES array.

Variable Attributes

The attributes in this section define form variables.[5] They are divided into two groups, those for creating and changing variables and those for returning the values of other variable attributes.

Creating and Changing Variables

Each attribute for creating or changing variables is specified as a value in a record in an initialized array. Each record is of type FDT$VARIABLE_ATTRIBUTE, which is listed in Appendix F.

Once established, this array is named on the VARIABLE_ATTRIBUTES parameter in the call to the FDP$CHANGE_VARIABLE or FDP$CREATE_VARIABLE procedure, described later in this chapter.

The following are the attribute records, their descriptions, and the permitted values for each. The attribute record names are in italics.

add_valid_integer_range,

delete_valid_integer_range

Adds or deletes a range of integer values that are valid for the variable. The range must not overlap any existing integer ranges. To specify more than one range, you may use this record more than once. The range specified for delete_valid_integer_range must correspond to the range that was specified by add_valid_integer_range. This record has two fields:

maximum_integer

The maximum integer value for the variable (type integer).

minimum_integer

The minimum integer value for the variable (type integer).

add_valid_real_range,

delete_valid_real_range

Adds or deletes a range of real values that are valid for the variable. The range must not overlap any existing real ranges. To specify more than one range, you may use this record more than

5. For more information on variables, refer to More About Forms, earlier in this chapter.

once. The range specified for delete_valid_real_range must correspond to the range that was specified by add_valid_real_range. This record has two fields:

maximum_real

The maximum real value for the variable (type real).

minimum_real

The minimum real value for the variable (type real).

add_valid_string,

delete_valid_string

Adds or deletes a string that is valid for the variable. To specify more than one string, you may use this record more than once. This record specifies a pointer (type ^FDT$VALID_STRING) to a string of characters which the user may enter at the terminal. Comparison takes place according to the rules laid down by the string_compare_rules attribute, described later in this section.

input_format

Specifies the data-entry format for the terminal. This is a variant record (type FDT$INPUT_FORMAT). Its KEY field (type FDT$INPUT_FORMAT_KEY) contains one of the following:

FDC$CHARACTER_INPUT_FORMAT

Allows any ASCII characters. This is the default value.

FDC$ALPHABETIC_INPUT_FORMAT

Allows alphabetic characters only (upper and lower case A through Z).

FDC$DIGITS_INPUT_FORMAT

Allows numeric characters only (0 through 9).

FDC$REAL_INPUT_FORMAT

Allows real numbers in the format of FORTRAN F, E, or G.

FDC$SIGNED_INPUT_FORMAT

Allows numeric characters with or without leading signs.

FDC$CURRENCY_INPUT_FORMAT

Allows numeric characters in a currency format, such as dollars-and-cents (type FDT$INPUT_CURRENCY_FORMAT). This record contains three fields, each as a one-character string: a currency symbol, a thousands separator, and a decimal point. The defaults are a dollar sign ($), a comma (,), and a period (.) respectively.

io_mode

Specifies the input and output transferring of variables (type FDT$IO_MODE). The following values are available:

FDC$PROGRAM_INPUT_OUTPUT

Programs save data from one application user interaction to another. The user does not see the entered variable.

FDC$TERMINAL_INPUT

The user inputs data, which is blanked out as soon as possible.

FDC$TERMINAL_INPUT_OUTPUT

The user inputs data, which remains visible. The program outputs data to this variable. This is the default value.

FDC$TERMINAL_OUTPUT

The program outputs data to the terminal (the user cannot enter data). Any modification of the variable is corrected as soon as possible.

new_variable_name

Specifies another name for a variable (type OST$NAME). The form processor language rules must be obeyed.

error_display

Specifies the attribute used for displaying an error when a variable does not pass validation. This record may contain one or more values from the following subset of the FDT$DISPLAY_ATTRIBUTE_SET. The default value is FDC$INVERSE_VIDEO.

FDC$INVERSE_VIDEO
FDC$LOW_INTENSITY
FDC$HIGH_INTENSITY
FDC$BLINK
FDC$UNDERLINE
FDC$BLACK_FOREGROUND
FDC$BLUE_FOREGROUND

FDC$GREEN_FOREGROUND
FDC$MAGENTA_FOREGROUND
FDC$RED_FOREGROUND
FDC$CYAN_FOREGROUND
FDC$YELLOW_FOREGROUND
FDC$WHITE_FOREGROUND
FDC$BLACK_BACKGROUND
FDC$BLUE_BACKGROUND
FDC$GREEN_BACKGROUND
FDC$MAGENTA_BACKGROUND
FDC$RED_BACKGROUND
FDC$CYAN_BACKGROUND
FDC$YELLOW_BACKGROUND
FDC$WHITE_BACKGROUND
FDC$ITALIC_DISPLAY_ATTRIBUTE
FDC$TITLE_DISPLAY_ATTRIBUTE
FDC$INPUT_DISPLAY_ATTRIBUTE
FDC$ERROR_DISPLAY_ATTRIBUTE
FDC$MESSAGE_DISPLAY_ATTRIBUTE

output_format

Contains a variant record (type FDT$OUTPUT_FORMAT) specifying the output format and the length of the formatted output for a variable text object.

Its KEY field (type FDT$OUTPUT_FORMAT_KEY) contains one of the following output formats:

FDC$CHARACTER_OUTPUT_FORMAT

The ASCII characters are output as is. This record specifies the character field width, which corresponds to the FORTRAN A descriptor.

FDC$CURRENCY_OUTPUT_FORMAT

Allows numeric characters in a currency format, such as dollars-and-cents (type FDT$OUTPUT_CURRENCY_FORMAT). This record contains six fields. The first three are one-character strings each: a currency symbol, a thousands separator, and a decimal point. The last three fields are the following:

field_width

Specifies the length of the entry including punctuation (type FDT$TEXT_LENGTH).

sign_treatment

Determines whether a sign is used (type FDT$SIGN_TREATMENT and MLT$SIGN_TREATMENT). The following values are available:

MLC$MINUS_IF_NEGATIVE

If the number is negative, it is prefixed with a minus sign; if it is positive, no sign is used.

MLC$ALWAYS_SIGNED

If the number is negative, it is prefixed with a minus sign, if positive, a plus sign.

suppress_leading_zeros

Type boolean. If TRUE, zero currency values are displayed as blanks.

FDC$E_E_OUTPUT_FORMAT, FDC$G_E_OUTPUT_FORMAT

These are the FORTRAN Ew.dEe and Gw.dEe formats (type FDT$EXPONENT_OUTPUT_FORMAT). This record contains the following fields:

field_width

The FORTRAN w descriptor (type FDT$REAL_FIELD_WIDTH).

digits_in_exponent

The FORTRAN e descriptor (type FDT$DIGITS_IN_EXPONENT).

digits_right_decimal

The FORTRAN d descriptor (type FDT$DIGITS_RIGHT_DECIMAL).

sign_treatment

A value of MLC$MINUS_IF_NEGATIVE or MLC$ALWAYS_SIGNED (type FDT$SIGN_TREATMENT). These are described under FDC$CURRENCY_OUTPUT_FORMAT, above.

suppress_zero

A boolean value. If TRUE, a zero is displayed as spaces.

FDCF_OUTPUT_FORMAT, FDCE_OUTPUT_FORMAT, FDC$G_OUTPUT_FORMAT

This record specifies the FORTRAN Fw.d, Ew.d, and Gw.d formats (type FDT$FLOAT_OUTPUT_FORMAT). It contains the following fields:

digits_right_of_decimal

The FORTRAN d descriptor (type FDT$DIGITS_RIGHT_DECIMAL).

field_width

The FORTRAN w descriptor (type FDT$REAL_FIELD_WIDTH).

sign_treatment

A value of MLC$MINUS_IF_NEGATIVE or MLC$ALWAYS_SIGNED (type FDT$SIGN_TREATMENT).

suppress_zero

A boolean. If TRUE, a zero is displayed as spaces.

FDC$INTEGER_OUTPUT_FORMAT

This record (type FDT$INTEGER_OUTPUT_FORMAT) corresponds to the FORTRAN I format. It contains the following fields:

field_width

The FORTRAN w descriptor (type FDT$INTEGER_FIELD_WIDTH).

minimum_output_digits

The FORTRAN m descriptor (type FDT$MINIMUM_OUTPUT_DIGITS).

sign_treatment

A value of MLC$MINUS_IF_NEGATIVE or MLC$ALWAYS_SIGNED (type FDT$SIGN_TREATMENT).

program_data_type

Specifies the program data type for the variable (type FDT$PROGRAM_DATA_TYPE) using one of the following values:

FDC$PROGRAM_CHARACTER_TYPE

The characters entered by the user are passed to the program.

FDC$PROGRAM_INTEGER_TYPE

The characters entered by the user are converted to an integer.

FDC$PROGRAM_REAL_TYPE

The characters entered by the user are converted to a real type.

FDC$PROGRAM_UPPER_CASE_TYPE

The characters entered by the user are converted to uppercase before being transferred to the program. The characters transferred by the program to the form are also converted to uppercase.

string_compare_rules

Specifies how the terminal input is compared to valid strings specified for the variable. For information on establishing valid strings for a variable, refer to the add_valid_string attribute earlier in this section. Contains two fields:

compare_in_upper_case

A boolean. If TRUE, the user's input is converted to upper case before the comparison is made with the valid strings. Otherwise, the user's input is not changed before the comparison is made.

compare_to_unique_substring

A boolean. If TRUE, the user may enter a unique substring for the value. The comparison starts at column 1. The complete strings are defined by the add_valid_string record. The application program gets the entire string as specified by add_valid_string.

terminal_user_entry

Specifies whether or not the user must make an entry (type FDT$TERMINAL_USER_ENTRY). The following values are available:

FDC$ENTRY_OPTIONAL

An entry is not required.

FDC$MUST_ENTER

An entry is required. If the user makes the entry, deletes the form, and then redisplays the form, the entry will not have to be made again. To require the reentry of the value, use the FDP$RESET_FORM procedure in addition to FDC$MUST_ENTER.

FDC$MAY_ENTER_UNKNOWN

Currently unused.

FDC$MUST_FILL

Currently unused.

unused_variable_entry

Indicates a null filler in the FDT$VARIABLE_ATTRIBUTES array.

variable_error

Contains a variant record (type FDT$ERROR_DEFINITION) specifying the error processing for the variable. Its KEY field (type FDT$ERROR_KEY) contains one of the following:

FDC$ERROR_FORM

The name of an application-defined form to be displayed (type OST$NAME).

FDC$ERROR_MESSAGE

A pointer to the message to be displayed (type ^FDT$ERROR_MESSAGE).

FDC$NO_ERROR_RESPONSE

Screen Formatting does not display an error form or message when the user enters invalid data. The application program must process the VARIABLE_STATUS parameter when getting variables.

variable_help

Contains a variant record (type FDT$HELP_DEFINITION) specifying the help information provided when the user executes a help event with the cursor placed on the variable. Its KEY field (type FDT$HELP_KEY) contains one of the following:

FDC$HELP_FORM

The name of an application-defined form containing the help (type OST$NAME).

FDC$HELP_MESSAGE

A pointer to a help message (type ^FDT$HELP_MESSAGE).

FDC$NO_HELP_RESPONSE

Screen Formatting does nothing when the user executes the help event.

variable_length

Contains an input field that specifies the character length of the data area for a character variable (type FDT$VARIABLE_LENGTH). If the length is not specified, the size of the screen text object for the variable is used. The user can execute scrolling commands to see all the data in the program variable. This attribute does not apply to real and integer data types.

Getting Variable Attributes

The following attribute records return additional attributes of a form. These records are specified in an initialized array. Each record is of type FDT$GET_VARIABLE_ATTRIBUTE, which is listed in Appendix F.

Once established, this array is named on the GET_VARIABLE_ATTRIBUTES parameter in the call to the FDP$GET_VARIABLE_ATTRIBUTES procedure, described later in this chapter.

The following are the attribute records, their descriptions, and the permitted values for each. The attribute record names are in italics.

get_error_display

Returns the display attribute(s) used when the variable does not pass validation (type FDT$DISPLAY_ATTRIBUTE_SET). For a list of the attributes, refer to the error_display attribute earlier in this chapter under *Creating and Changing Variables.*

get_input_format

Contains a variable record (type FDT$INPUT_FORMAT) that returns the type of data the user can enter. For a list of the types, refer to the input_format attribute earlier in this chapter under *Creating and Changing Variables.*

get_io_mode

Returns the input and output transfers done for the variable (type FDT$IO_MODE). For a description of the values, refer the io_mode attribute earlier in this chapter under *Creating and Changing Variables.*

get_next_valid_real_range

Returns the next range of real values that are valid for the variable. To return more than one range, you can use this record more than once. The first record returns the first range, the second record returns the second range, and so on.

This record contains two fields:

minimum_real

The minimum real valid value for the variable (type real).

maximum_real

The maximum real valid value for the variable (type real).

get_next_valid_string

Returns to the pointer the next string of characters valid for the variable (type ^FDT$VALID_STRING). These are the characters the user can enter. To return more than one string, you can use this record more than once. The first record returns the first string, the second record returns the second string, and so on.

get_number_valid_integers

Returns the number of valid integer ranges (type FDT$NUMBER_VALID_INTEGERS). You then allocate an array of attributes to get the valid integer ranges and use the get_valid_integer_range attribute to return them.

get_number_valid_reals

Returns the number of valid real ranges (type FDT$NUMBER_VALID_REALS). You then allocate an array of attributes to get the valid real ranges and use the get_next_valid_real_range attribute to return them.

get_number_valid_strings

Returns the number of valid strings (type FDT$NUMBER_VALID_STRINGS). You then allocate an array of attributes to get the lengths of the valid strings and use the get_next_valid_string attribute to return them.

get_output_format

Returns the output format (type FDT$OUTPUT_FORMAT). For a description of this record, refer to the output_format attribute earlier in this chapter under *Creating and Changing Variables.*

get_program_data_type

Returns the data type the program uses for manipulation (type FDT$PROGRAM_DATA_TYPE). For a description of this record, refer to the description of the program_data_type attribute earlier in this chapter under *Creating and Changing Variables.*

get_string_compare_rules

Returns the values that specify how the terminal input is compared to valid strings specified for the variable. Contains the fields compare_in_upper_case and compare_to_unique_substring. For a description of these fields, refer to the string_compare_rules attribute earlier in this chapter under *Creating and Changing Variables.*

get_unused_variable_entry

Indicates a null filler in the FDT$GET_VARIABLE_ATTRIBUTES array.

get_valid_integer_range

Returns the next range of integer values that are valid for the variable. To return more than one range, you may use this record more than once. The first record returns the first range, the second record returns the second range, and so forth.

This record contains two fields:

minimum_integer

The minimum integer value valid for the variable (type integer).

maximum_integer

The maximum integer value valid for the variable (type integer).

get_valid_string_length

Returns the length of a string for valid string validation (type FDT$VALID_STRING_LENGTH). To return more than one string length, you can use this record more than once. The first record returns the first valid string length, the second record returns the second length, and so forth.

get_var_error_message

Contains an input field that returns the message displayed in the message form when the data entered by the user does not pass validation (type ^FDT$ERROR_MESSAGE). The error message is returned to the string specified by this pointer.

get_var_help_message

Contains an input field that returns the message displayed in the message form when the user executes the help event on this variable (type ^FDT$HELP_MESSAGE). The help message is returned to the string specified by this pointer.

get_variable_error

Contains a variant record that returns information about error processing for the variable (type FDT$GET_ERROR_DEFINITION). Its KEY field (type FDT$GET_ERROR_KEY) contains one of the following:

FDC$GET_ERROR_FORM

The name of the error form (type OST$NAME).

FDC$GET_ERROR_MESSAGE

The length of the error message in characters (type FDT$ERROR_MESSAGE_LENGTH). You then allocate a string of this length and use the FDP$GET_VARIABLE_ATTRIBUTES procedure with the get_var_error_message record to obtain the message.

FDC$GET_NO_ERROR_RESPONSE

Screen Formatting does not display an error form or message when the user enters invalid data.

get_variable_help

Contains a variant record that returns information about help processing for the variable (type FDT$GET_HELP_DEFINITION). This processing applies when the user executes the help event with the cursor placed on the variable. Its KEY field (type FDT$GET_HELP_KEY) contains one of the following:

FDC$GET_HELP_FORM

The name of the help form (type OST$NAME).

FDC$GET_HELP_MESSAGE

The length of the help message in characters (type FDT$HELP_MESSAGE_LENGTH). You then allocate a string of this length and use the FDP$GET_VARIABLE_ATTRIBUTES procedure with the get_var_help_message attribute to obtain the message.

FDC$GET_NO_HELP_RESPONSE

Screen Formatting does not display a help form or message.

get_variable_length

Returns the character length of the program data area for the variable (type FDT$VARIABLE_LENGTH).

Table Attributes

The attributes in this section describe the tables containing variables. These attributes are divided into two groups, those for creating and changing tables and those for returning the values of other table attributes.

Creating and Changing Tables

Each attribute for creating or changing tables is specified as a value in a record in an initialized array. Each record is of type FDT$TABLE_ATTRIBUTE, which is listed in Appendix F.

Once established, this array is named on the TABLE_ATTRIBUTES parameter in the call to the FDP$CHANGE_TABLE or FDP$CREATE_TABLE procedure, described later in this chapter.

The following are the attribute records, their descriptions, and the permitted values for each. The attribute record names are in italics.

add_table_variable,

delete_table_variable

Associates a variable with a table or deletes one from a table (type OST$NAME).

For add_table_variable, the following rules apply:

- The variable name can already have been created when this attribute is specified.
- The variable name must exist when the form definition ends, but must not currently exist in the list of variable names associated with the table.
- The name must obey the rules for names given by the form processor.
- A variable cannot be associated with more than one table.

For delete_table_variable, any variable definition created by the FDP$CREATE_VARIABLE procedure is not deleted.

new_table_name

Specifies a new name for the table (type OST$NAME). The name must follow the rules for names given by the form processor language. The new name must be unique.

stored_occurrence

Specifies the maximum number of stored occurrences allowed in the table (type FDT$OCCURRENCE). The value must be greater than or equal to the value for the visible_occurrence attribute, described below. The default value is 1. (You can create stored objects using FDP$CREATE_STORED_OBJECT.)

unused_table_entry

Indicates a null filler in the FDT$TABLE_ATTRIBUTES array.

visible_occurrence

Specifies the number of occurrences in the table that are visible to the user (type FDT$OCCURRENCE). You must create a visible object that is variable text for each occurrence on the form (FDP$CREATE_OBJECT).

This attribute is optional. The default is the current value of the stored_occurrence attribute (described above).

Getting Table Attributes

The following records return the values of other table attributes. These records are specified in an initialized array. Each record is of type FDT$GET_TABLE_ATTRIBUTE, which is listed in Appendix F.

Once established, this array is named on the GET_TABLE_ATTRIBUTES parameter in the call to the FDP$GET_TABLE_ATTRIBUTES procedure, described later in this chapter.

The following are the attribute records, their descriptions, and the permitted values for each. The attribute record names are in italics.

get_next_table_variable

Returns the next variable associated with the table (type OST$NAME). To return more than one variable, you can use this record more than once. The first record in the array returns the first variable, the second returns the second variable, etc.

get_number_table_variables

Returns the number of variables in the table (type FDT$NUMBER_TABLE_VARIABLES). You can use this record to allocate an array and then use the get_next_table_variable attribute to return the variables.

get_stored_occurrence

Returns the number of stored occurrences in the table (type FDT$OCCURRENCE).

get_unused_table_entry

Indicates a null filler in the FDT$GET_TABLE_ATTRIBUTES array.

get_visible_occurrence

Returns the number of occurrences in the table that are visible to the user (type FDT$OCCURRENCE).

Form Definition Record Attributes

The attributes in this section are in two groups, those for creating and changing form definition records and those for getting other form definition record attributes.

Changing Record Attributes

Each attribute for creating or changing form definition records is specified as a value in a record in an initialized array. Each record is of type FDT$RECORD_ATTRIBUTE, which is listed in Appendix F.

Once established, this array is named on the RECORD_ATTRIBUTES parameter in the call to the FDP$CHANGE_FORM_RECORD procedure, described later in this chapter.

The following are the attribute records, their descriptions, and the permitted values for each. The attribute record names are in italics.

record_deck_name

Specifies the Source Code Utility deck name for the form definition record (type OST$NAME). If you don't specify this name, the form name is used.

record_name

Specifies the name of the record (type OST$NAME). This is the form definition record name. In COBOL, the name is a COBOL 01-level data name; in Pascal, SCL, and CYBIL, it is a record type name; in FORTRAN, it is name defined in a CHARACTER statement that specifies the storage for the form.

If you don't specify this name, the deck name is used.

unused_table_entry

Indicates a null filler in the FDT$RECORD_ATTRIBUTES array.

Getting Record Attributes

The following records return the values of other record attributes. These records are specified in an initialized array. Each record is of type FDT$GET_RECORD_ATTRIBUTE, which is listed in Appendix F.

Once established, this array is named on the GET_RECORD_ATTRIBUTES parameter in the call to the FDP$GET_RECORD_ATTRIBUTES procedure, described later in this chapter.

The following are the attribute records, their descriptions, and the permitted values for each. The attribute record names are in italics.

get_record_deck_name

Returns the name of the deck for the Source Code Utility (type OST$NAME).

get_record_length

Returns the length of the record in cells (type FDT$RECORD_LENGTH).

get_record_name

Returns the record name (type OST$NAME).

get_unused_record_entry

Indicates a null filler in the FDT$GET_RECORD_ATTRIBUTES array.

Object Attributes

This section describes the attributes for form objects.[6] Objects can be either text or graphics. These attributes are in two groups, those for creating and changing objects and those for returning the values of other object attributes.

Creating and Changing Objects

Each attribute for creating or changing an object is specified as a value in a record in an initialized array. Each record is of type FDT$OBJECT_ATTRIBUTE, which is listed in Appendix F.

Once established, this array is named on the OBJECT_ATTRIBUTES parameter in the call to the FDP$CHANGE_OBJECT or FDP$CREATE_OBJECT procedure, described later in this chapter.

The following are the attribute records, their descriptions, and the permitted values for each. The attribute record names are in italics.

object_display

Specifies a set of display attributes for the object (type FDT$DISPLAY_ATTRIBUTE_SET). When the object is displayed, this attribute is used. For a list of these attributes, refer to the add_display_definition attribute under *Basic Form Attributes* earlier in this chapter.

object_height

Specifies the height of the object (type FDT$HEIGHT).

object_line_x_increment

Specifies the new x increment by changing the x increment from the line origin to the line destination (type FDT$X_INCREMENT).

object_line_y_increment

Specifies the new y increment by changing the y increment from the line origin to the line destination (type FDT$Y_INCREMENT).

6. For more information on objects, refer to chapter 1.

object_name

Specifies a name for the object. The object name must follow the conventions of the form processor. Use the object name to associate an object on the form with a variable definition. This record contains two fields:

object_name

The name of the object (type OST$NAME).

occurrence

The occurrence of the name (type FDT$OCCURRENCE).

object_position

Specifies a new position for the object, with the following two fields:

x_position

The new x position (type FDT$X_POSITION).

y_position

The new y position (type FDT$Y_POSITION).

object_text

Changes the text associated with an object or a constant text box object. Specifies a pointer to the new text (type ^FDT$TEXT).

object_text_processing

Changes the text processing for a text box object (type FDT$TEXT_BOX_PROCESSING). Contains the following values:

FDC$CENTER_CHARACTERS

Currently unused.

FDC$WRAP_CHARACTERS

Wraps data that extends past the left boundary of the box onto the next line, character-by-character.

FDC$WRAP_WORDS

Wraps data at the left boundary of the box onto the next line, word-by-word. A space indicates the end of a word.

object_width

Specifies the width of the object (type FDT$WIDTH).

unused_object_entry

Indicates a null filler in the FDT$OBJECT_ATTRIBUTES array.

Getting Object Attributes

The following records return the values of other object attributes. These records are specified in an initialized array. Each record is of type FDT$GET_OBJECT_ATTRIBUTE, which is listed in Appendix F.

Once established, this array is named on the OBJECT_ATTRIBUTES parameter in the call to the FDP$GET_OBJECT_ATTRIBUTES procedure, described later in this chapter.

The following are the attribute records, their descriptions, and the permitted values for each. The attribute record names are in italics.

get_object_definition

Returns the object definition. This is a variant record (type aDT$GET_OBJECT_DEFINITION). Its KEY field (type FDT$OBJECT_DEFINITION_KEY) can contain one of the following values. (For each of these values, additional fields describe each object.)

FDC$BOX

Describes the box with two fields:

box_width

The character width (1 .. FDC$MAXIMUM_X_POSITION) of the box (type FDT$WIDTH).

box_height

The character height (1 .. FDC$MAXIMUM_Y_POSITION) of the box (type FDT$HEIGHT).

FDC$LINE

Describes the line with two fields:

x_increment

The number of characters needed to increment the x line origin position given in the request to determine the end point of the line (type FDT$X_INCREMENT).

y_increment

The number of characters needed to increment the y line origin position given in the request to determine the end point of the line (type FDT$Y_INCREMENT).

FDC$CONSTANT_TEXT

Displays constant text on the form. Contains two fields:

constant_text_width

The width of the constant text in characters on the screen (type FDT$WIDTH).

constant_text_length

The length of the text in characters (type FDT$TEXT_LENGTH). Use the get_object_text attribute to obtain the text. The text length indicates how much space is needed to hold the text.

FDC$CONSTANT_TEXT_BOX

Describes a constant text box on the form. The text can occupy several lines. Contains four fields:

constant_box_height

The height of the text area in characters (type FDT$HEIGHT).

constant_box_processing

Type FDT$TEXT_BOX_PROCESSING. Refer to object_text_processing under *Creating and Changing Objects.*

constant_box_width

The width of the text area in characters (type FDT$WIDTH).

constant_box_text_length

The number of characters of text created for the text box (type FDT$TEXT_LENGTH). Allocate the amount of space needed for the text using the text length, then use the get_object_text attribute.

FDC$TABLE

Currently unused.

FDC$VARIABLE_TEXT_BOX

Describes a variable text box on the form. The text can occupy more than one line. Contains four fields:

variable_box_height

The height of the text area in characters (type FDT$HEIGHT).

variable_box_processing

Type FDT$TEXT_BOX_PROCESSING. Refer to object_text_processing under *Creating and Changing Objects.*

variable_box_text_length

The number of characters of text created for the text box (type FDT$TEXT_LENGTH). Allocate the amount of space needed for the text using the text length, then use the get_object_text attribute.

variable_box_width

The width of the text area in characters (type FDT$WIDTH).

FDC$VARIABLE_TEXT

Describes a variable text object on the form. Contains two fields:

variable_text_length

The number of characters of text created for the variable text (type FDT$TEXT_LENGTH). Allocate the amount of space needed for the text using the text length, then use the get_object_text attribute.

variable_text_width

The visible form width of the text area in characters (type FDT$WIDTH).

get_object_display

Returns the display attribute for the object (type FDT$DISPLAY_ATTRIBUTE_SET). For a list of these attributes, refer to the add_display_definition attribute under *Basic Form Attributes* earlier in this chapter.

get_object_name

Returns the name for the object. For a description of this attribute, refer to object_name under *Creating and Changing Objects.*

get_object_text

Returns the object text to the specified pointer (type ^FDT$TEXT).

get_object_text_length

Returns the character length of the text (type FDT$TEXT_LENGTH).

get_unused_object_entry

Indicates a null filler in the FDT$GET_OBJECT_ATTRIBUTES array.

CYBIL Screen Formatting Procedures

Use the following CYBIL procedure calls when creating forms within a CYBIL program.

Changing a Form

Purpose FDP$CHANGE_FORM procedure changes the attributes that apply to the entire form.

Format **FDP$CHANGE_FORM (form_identifier, form_attributes, status)**

Parameters form_identifier: fdt$form_identifier;

The form identifier established when the form was opened.

form_attributes: VAR { input-output } of fdt$form_attributes;

An array containing form attributes.

status: VAR of ost$status;

The status variable in which the completion status is returned.

Conditions

fde$bad_data_value
fde$cannot_update_opened_form
fde$display_name_exists
fde$event_name_exists
fde$invalid_display_name
fde$invalid_event_name
fde$invalid_form_area_key
fde$invalid_form_identifier
fde$invalid_form_language
fde$invalid_form_name
fde$no_comments_to_delete
fde$no_space_available
fde$system_error
fde$unknown_display_name
fde$unknown_event_name

Changing the Form Definition Record

Purpose

FDP$CHANGE_FORM_RECORD procedure changes the form definition record used to transfer data from the program to Screen Formatting, and from Screen Formatting to the program.

Format

FDP$CHANGE_FORM_RECORD (form_identifier, record_attributes, status)

Parameters

form_identifier: fdt$form_identifier;

The form identifier established when the form was opened.

record_attributes: VAR { input-output } of fdt$record_attributes;

An array containing record attributes.

status: VAR of ost$status;

The status variable in which the completion status is returned.

Conditions

fde$cannot_update_opened_form
fde$invalid_form_identifier
fde$invalid_deck_name
fde$invalid_record_name
fde$invalid_table_name

Changing an Object

Purpose FDP$CHANGE_OBJECT procedure changes the object attributes.

Format **FDP$CHANGE_OBJECT (form_identifier, x_position, y_position, object_attributes, status)**

Parameters

form_identifier: fdt$form_identifier;

The form identifier established when the form was opened.

x_position: fdt$x_position;

The x position of the object relative to the form.

y_position: fdt$y_position;

The y position of the object relative to the form.

object_attributes: VAR { input-output } of fdt$object_attributes;

An array of object attributes.

status: VAR of ost$status;

The status variable in which the completion status is returned.

Conditions

fde$bad_data_value
fde$cannot_update_opened_form
fde$invalid_form_identifier
fde$invalid_object_change
fde$invalid_object_name
fde$no_object_at_position
fde$no_space_available
fde$no_string_specified
fde$object_occurrence_exists
fde$system_error
fde$unknown_object_name

Changing a Stored Object

Purpose

FDP$CHANGE_STORED_OBJECT procedure changes the initial value for the occurrence of a table variable that does not initially appear on a form.

Format

FDP$CHANGE_STORED_OBJECT (form_identifier, name, occurrence, text, display_attribute_set, status)

Parameters

form_identifier: fdt$form_identifier;

The form identifier established when the form was opened.

name: ost$name;

The object name.

occurrence: fdt$occurrence;

The occurrence of the object.

text: fdt$text;

The text indicating the initial value.

display_attribute_set: fdt$display_attribute_set;

The set of attributes that describe how to display the object.

status: VAR of ost$status;

The status variable in which the completion status is returned.

Conditions

fde$bad_data_value
fde$cannot_update_opened_form
fde$invalid_form_identifier
fde$invalid_object_name
fde$invalid_occurrence
fde$no_space_available
fde$no_string_specified
fde$system_error
fde$unknown_object_name

Remarks

The user can see stored occurrences by executing paging or scrolling events.

Changing a Table

Purpose

FDP$CHANGE_TABLE procedure changes the attributes of a table.

Format

FDP$CHANGE_TABLE (form_identifier, table_name, table_attributes, status)

Parameters

form_identifier: fdt$form_identifier;

The form identifier established when the form was opened.

table_name: ost$name;

The name of the table.

table_attributes: VAR { input-output } of fdt$table_attributes;

An array containing table attributes.

status: VAR of ost$status;

The status variable in which the completion status is returned.

Conditions

fde$bad_data_value
fde$cannot_change_form
fde$invalid_form_identifier
fde$invalid_occurrence
fde$invalid_table_name
fde$invalid_variable_name
fde$no_space_available
fde$system_error
fde$table_name_exists
fde$unknown_table_name
fde$unknown_variable_name

Changing a Variable

Purpose FDP$CHANGE_VARIABLE procedure changes the variable attributes.

Format **FDP$CHANGE_VARIABLE (form_identifier, variable_name, variable_attributes, status)**

Parameters form_identifier: fdt$form_identifier;

The form identifier established when the form was opened.

variable_name: ost$name;

The variable name.

variable_attributes: VAR { input-output } of fdt$variable_attributes;

An array containing variable attributes.

status: VAR of ost$status;

The status variable in which the completion status is returned.

Conditions

fde$variable_name_exists
fde$valid_string_exists
fde$unknown_variable_name
fde$unknown_valid_string
fde$unknown_real_range
fde$unknown_integer_range
fde$system_error
fde$range_overlap
fde$no_string_specified
fde$no_space_available
fde$no_comments_to_delete
fde$invalid_variable_name
fde$invalid_real_range
fde$invalid_integer_range
fde$invalid_form_name
fde$invalid_form_identifier
fde$cannot_update_opened_form
fde$bad_data_value

Converting to Program Variable

Purpose FDP$CONVERT_TO_PROGRAM_VARIABLE procedure converts data entered by an application user to program data.

Format **FDP$CONVERT_TO_PROGRAM_VARIABLE (program_data_type, p_program_variable, program_variable_length, input_format, p_screen_variable, screen_variable_length, variable_status, status)**

Parameters program_data_type: fdt$program_data_type;

The variable definition of the data type the program uses to manipulate the variable.

p_program_variable: ^cell;

A pointer to the first cell to receive the converted data for the program variable.

program_variable_length: fdt$program_variable_length;

Length of the program variable in cells.

input_format: fdt$input_format;

The variable definition for the application user's input format.

p_screen_variable: ^fdt$text;

A pointer to the string that contains the characters entered by the application user to be converted.

screen_variable_length: fdt$text_length;

Length of the string containing the user's characters.

variable_status: VAR of fdt$variable status;

An ordinal value that gives you the status of the variable.

FDC$INVALID_BDP_DATA

The screen variable contains characters that can not be converted to the program data type.

FDC$LOSS_OF_SIGNIFICANCE

The screen variable is too large to fit in the program variable.

FDC$NO_ERROR

No error occurred on the conversion.

FDC$OVERFLOW

The screen variable when converted to the program variable is infinite or indefinite.

status: VAR of ost$status;

The status variable in which the completion status is returned.

Conditions fde$bad_data_value

Converting to Screen Variable

Purpose FDP$CONVERT_TO_SCREEN_VARIABLE procedure converts program data to characters for screen display.

Format **FDP$CONVERT_TO_SCREEN_VARIABLE (program_data_type, p_program_variable, program_variable_length, output_format, p_screen_variable, screen_variable_length, variable_status, status)**

Parameters program_data_type: fdt$program_data_type;

The variable definition of the data type the program uses to manipulate the variable.

p_program_variable: ^cell;

A pointer to the first cell of the program variable to be converted to the screen variable.

program_variable_length: fdt$program_variable_length;

Length of the program variable in cells.

output_format: fdt$output_format;

The variable definition for the screen output format.

p_screen_variable: ^fdt$text;

A pointer to the string to receive the characters converted from the program variable.

screen_variable_length: fdt$text_length;

Length of the string displayed at the user's screen.

variable_status: VAR of fdt$variable status;

An ordinal value that gives you the status of the variable.

FDC$BAD_PARAMETERS

The output format is not correct.

FDC$INDEFINITE

The program variable contains an indefinite number.

FDC$INVALID_BDP_DATA

The program variable contains characters used for terminal control.

FDC$LOSS_OF_SIGNIFICANCE

The program variable is too large to display in the area specified for screen display.

FDC$NO_ERROR

No error occurred on the conversion.

status: VAR of ost$status;

The status variable in which the completion status is returned.

Conditions fde$bad_data_value

Copying an Area

Purpose FDP$COPY_AREA procedure copies all objects and unprotected text from one area to another on a form.

Format **FDP$COPY_AREA (form_identifier, form_x_position, form_y_position, width, height, to_x_postion, to_y_position, status)**

Parameters form_identifier: fdt$form_identifier;

The form identifier established when the form was opened.

form_x_position: fdt$x_position;

The form x position of the area that encloses the data to be copied. The origin of the area is the upper left corner, relative to the form.

form_y_position: fdt$y_position;

The form y position of the area that encloses the data to be copied. The origin of the area is the upper left corner, relative to the form.

height: fdt$height;

The height of the area to be copied.

width: fdt$width;

The width of the area to be copied.

to_x_position: fdt$x_position;

The x position of the destination area upper left corner, relative to the form.

to_y_position: fdt$y_position;

The y position of the destination area upper left corner, relative to the form.

status: VAR of ost$status;

The status variable in which the completion status is returned.

Conditions

fde$area_cuts_object
fde$bad_data_value
fde$copy_outside_form
fde$invalid_form_identifier
fde$no_space_available
fde$object_overlays
fde$system_error

Remarks

- A design form has objects (protected text, line drawings) and unprotected text.

- A target form contains only objects. The area to be copied must not slice any objects. The destination area must not contain any objects. Object names and occurrence attributes are not copied.

Copying a Form

Purpose FDP$COPY_FORM procedure copies a form and assigns a new form identifier to the copied form.

Format **FDP$COPY_FORM (from_form_identifier, to_form_identifier, status)**

Parameters from_form_identifier: fdt$form_identifier;

The form identifier established when the form was opened.

to_form_identifier: VAR of fdt$form_identifier;

The new form identifier that Screen Formatting assigns to the copied form.

status: VAR of ost$status;

The status variable in which the completion status is returned.

Conditions
fde$bad_data_value
fde$invalid_form_identifier
fde$no_space_available
fde$system_error

Remarks To modify a copied form, you must issue an FDP$EDIT_FORM procedure.

Creating Constant Text

Purpose

FDP$CREATE_CONSTANT_TEXT procedure creates constant text objects for a target form using the unprotected text on the design form. These objects have the background and foreground attributes of the target form.

Format

FDP$CREATE_CONSTANT_TEXT (design_form_identifier, target_form_identifier, status)

Parameters

design_form_identifier: fdt$form_identifier;

The form identifier of a design form that Screen Formatting uses to create constant text objects.

target_form_identifier: fdt$form_identifier;

The form identifier of a target form where Screen Formatting stores the constant text objects.

status: VAR of ost$status;

The status variable in which the completion status is returned.

Conditions

fde$bad_data_value
fde$invalid_form_identifier
fde$no_space_available
fde$system_error

Creating a Design Form

Purpose FDP$CREATE_DESIGN_FORM procedure creates a form for designing other forms interactively.

Format **FDP$CREATE_DESIGN_FORM (form_identifier, form_attributes, status)**

Parameters form_identifier: VAR of fdt$form_identifier;

The form identifier established when the form was opened.

form_attributes: VAR { input-output } of fdt$form_attributes;

An array containing attributes that apply to the entire form.

status: VAR of ost$status;

The status variable in which the completion status is returned.

Conditions

fde$bad_data_value
fde$display_name_exists
fde$event_name_exists
fde$invalid_display_name
fde$invalid_event_name
fde$invalid_form_area_key
fde$invalid_form_identifier
fde$invalid_form_language
fde$invalid_form_name
fde$no_comments_to_delete
fde$no_space_available
fde$system_error
fde$terminal_disconnected
fde$unknown_display_name
fde$unknown_event_name

Remarks

- It is not necessary to execute an FDP$END_FORM procedure to indicate the end of the definition. You open the design form with the FDP$OPEN_FORM procedure and can then execute other procedures, such as FDP$ADD_FORM and FDP$READ_FORM.

- A table and a variable are created for the design form so the FDP$GET_STRING_VARIABLE and FDP$REPLACE_STRING_VARIABLE procedures can access text on the form. The variable character field width is the form width. Refer to Defining Attributes for a Form in this chapter to read about the attribute FDC$DESIGN_VARIABLE_NAME.

- The table has as many occurrences as the height of the form. On the design form, you can create constant objects and line drawing objects, but no variable objects.

Creating Design Text

Purpose

FDP$CREATE_DESIGN_TEXT procedure creates objects and unprotected text on the design form from objects defined on the target form.

Format

FDP$CREATE_DESIGN_TEXT (target_form_identifier, design_form_identifier, status)

Parameters

target_form_identifier: fdt$form_identifier;

The target form identifier to use as the source of the text for the design form.

design_form_identifier: fdt$form_identifier;

The form identifier of the design form.

status: VAR of ost$status;

The status variable in which the completion status is returned.

Conditions

fde$bad_data_value
fde$cannot_change_form
fde$invalid_design_form
fde$invalid_form_identifier
fde$no_space_available
fde$system_error

Remarks

The constant text objects on the target form (except for ones with the same color attributes as the target form) are created as objects on the design form. Constant text objects on the target form with the same color attributes as the target form and without names become free text on the design form.

Creating a Form

Purpose FDP$CREATE_FORM procedure creates a form.

Format **FDP$CREATE_FORM (form_identifier, form_attributes, status)**

Parameters form_identifier: VAR of fdt$form_identifier;

The form identifier established when the form was opened.

form_attributes: VAR { input-output } of fdt$form_attributes;

An array containing attributes that apply to the entire form.

status: VAR of ost$status;

The status variable in which the completion status is returned.

Conditions

fde$bad_data_value
fde$display_name_exists
fde$event_name_exists
fde$invalid_display_name
fde$invalid_event_name
fde$invalid_form_area_key
fde$invalid_form_identifier
fde$invalid_form_language
fde$invalid_form_name
fde$no_comments_to_delete
fde$no_space_available
fde$system_error
fde$unknown_display_name
fde$unknown_event_name

Remarks After creating a form, you must issue an FDP$END_FORM procedure to end it.

Creating an Event Form

Purpose FDP$CREATE_EVENT_FORM procedure creates a form to display events.

Format **FDP$CREATE_EVENT_FORM (event_menus, form_attributes, form_identifier, status)**

Parameters event_menus: array [1 ..*];

An array of fdt$event_menu records. This record contains three fields:

event_label

The initial variable value on the form for the variable event_name.

event_name

The event name the application program uses to recognize the event. Also the variable name an application program can use to change the display attribute or event label value.

event_trigger

The event trigger on the terminal that causes the event.

form_attributes: VAR { input-output } of fdt$form_attibutes;

An array containing attributes that apply to the entire form. Screen Formatting calculates the form size based on the number of application event triggers given in the event_menus. Screen Formatting calculates the form location based on the form size and home cursor position of the terminal.

If the home cursor position is on the first line of the terminal screen, the event form occupies the last line of the terminal. If the home cursor position is on the last line of the terminal, then the event form occupies the next to the last line of the terminal.

form_identifier: VAR of fdt$form_identifier;

The form identifier established when the form was opened.

status: VAR of ost$status;

The status variable in which the completion status is returned.

Conditions

fde$bad_data_value
fde$invalid_display_name
fde$invalid_event_name
fde$invalid_form_language
fde$invalid_form_name
fde$no_space_available
fde$system_error
fde$unknown_event_name

Remarks

- This procedure ends the event form definition.
- The form identifier the procedure returns was established when the form was opened.
- Save the form by executing the FDP$WRITE_FORM_DEFINITION procedure.

Creating a Mark

Purpose FDP$CREATE_MARK procedure creates a display attribute for a specific text area.

Format **FDP$CREATE_MARK (form_identifier, start_x_position, start_y_position, end_x_position, end_y_position, status)**

Parameters

form_identifier: fdt$form_identifier;

The form identifier established when the form was opened.

start_x_position: fdt$x_position;

The form x position that starts the mark on the form. The leftmost character is 1. X positions go from left to right.

start_y_position: fdt$x_position;

The form y position that starts the mark on the form. The top character is 1. Y positions go from top to bottom.

end_x_position: fdt$x_position;

The end x position to end the mark.

end_y_position: fdt$x_position;

The end y position to end the mark.

status: VAR of ost$status;

The status variable in which the completion status is returned.

Conditions

fde$area_cuts_object
fde$bad_data_value
fde$create_mark_invalid
fde$form_not_scheduled
fde$form_pushed
fde$mark_outside_form
fde$no_space_available
fde$system_error

Remarks

- The marked area of the form must not slice any objects.
- This attribute marks text on which the terminal user wants to perform an operation.
- This procedure applies only to a design form.

Creating an Object

Purpose FDP$CREATE_OBJECT procedure creates an object on the form.

Format **FDP$CREATE_OBJECT (form_identifier, x_position, y_position, object_definition, object_attributes, status)**

Parameters form_identifier: fdt$form_identifier;

The form identifier established when the form was opened.

x_position: fdt$x_position;

The x coordinate for the origin of the object relative to the form.

y_position: fdt$x_position;

The y coordinate for the origin of the object relative to the form.

object_definition: fdt$object_definition;

This is a variant record that specifies the object type. Its KEY fields contain the following:

FDC$BOX

Draws a box on the form image. The FDP$CREATE_OBJECT procedure gives the origin of the box with respect to the origin of the form. The origin of the form is the upper left corner. The origin of the box is the upper left corner. The FDC$BOX field contains two other fields:

box_width

The width, in characters, of the box (type FDT$WIDTH). The box_width must be greater than, or equal to, one.

box_height

The height, in characters, of the box (type FDT$HEIGHT). The box_height must be greater than, or equal to, one.

FDC$CONSTANT_TEXT

Displays constant text on the form image. The origin of the text object is given by the FDP$CREATE_OBJECT procedure. The text can occupy all or part of a row on the form. The FDC$CONSTANT_TEXT field contains two other fields:

constant_text_width

The width of the constant text, in characters, on the screen (type FDT$WIDTH). This must be a number greater than or equal to one.

p_constant_text

This is the text to display on the form image (type ^FDT$TEXT).

FDC$LINE

Draws a line on the form image. The FDC$LINE field contains two other fields:

x_increment

The number of characters to increment the x position given in the procedure to determine the end point of the line (type FDT$X_INCREMENT). Some terminals only support vertical and horizontal lines. The x increment can be greater than or equal to zero.

y_increment

The number of characters to increment the y position given in the procedure to determine the end point of the line (type FDT$Y_INCREMENT). The x increment can greater than or equal to zero.

FDC$CONSTANT_TEXT_BOX

Defines an area on the form to display constant text. The text can occupy several lines. You can specify how text that crosses the right boundary of the text box is processed. The FDC$CONSTANT_TEXT_BOX field contains the following fields:

constant_text_box_height

The height of the text area, in characters (type FDT$HEIGHT). This must be greater than, or equal to, one.

constant_text_box_processing

Uses FDC$WRAP_WORD to wrap data at the right boundary of the box to the next line, if any, on a word basis (type FDT$TEXT_BOX_PROCESSING). The text for word wrap processing can include a formatting character. FDC$NEW_LINE_CHARACTER causes Screen Formatting to start a new line in a text box. Uses FDC$WRAP_CHARACTER to wrap data that goes past the right boundary of the box to the next line on a character basis.

constant_text_box_width

The width of the text area, in characters (type FDT$WIDTH). This must be greater than, or equal to, one.

p_constant_box_text

The text to display on the form image (type ^FDT$TEXT).

FDC$VARIABLE_TEXT

Defines an object for variable text. You associate the object to the variable by using an object name specified through the object attributes. The FDC$VARIABLE_TEXT field contains the following fields:

p_variable_text

The pointer to the text (type ^FDT$TEXT). This is the initial value of the variable.

variable_text_width

The width of the variable text in characters on the screen (type FDT$WIDTH). This must be a number greater than or equal to one.

FDC$VARIABLE_TEXT_BOX

Defines an area on the form to display variable text. The text can occupy several lines. You associate the object to the variable by using an object name specified through the object attributes. You can specify how text that crosses the right boundary of the text box is processed. The FDC$VARIABLE_TEXT_BOX field contains the following fields:

p_variable_box_text

The pointer to the text (type ^FDT$TEXT). This is the initial value of the variable.

variable_text_box_height

The height of the text area, in rows (type FDT$HEIGHT). This must be a number greater than or equal to one.

variable_text_box_processing

Uses FDC$WRAP_WORD to wrap data at the right boundary of the box to the next line, if any, on a word basis (type FDT$TEXT_BOX_PROCESSING). The text for word wrap processing can include a formatting character. The FDC$NEW_LINE_CHARACTER causes Screen Formatting to start a new line in a text box. Uses FDC$WRAP_CHARACTER to wrap data that goes past the right boundary of the box to the next line on a character basis.

variable_text_box_width

The width of the text area, in characters (type FDT$WIDTH). This must be greater than or equal to one.

object_attributes: VAR { input-output } of fdt$object_attributes;

An array of object attributes.

status: VAR of ost$status;

The status variable in which the completion status is returned.

Conditions

fde$bad_data_value
fde$cannot_update_opened_form
fde$invalid_form_identifier
fde$invalid_object_change
fde$invalid_object_name
fde$no_space_available
fde$no_string_specified
fde$object_occurrence_exists
fde$system_error

Remarks

- When the program data is too large to display on the form, it can be put into a text box. The user can then execute scroll events to see or modify the text.

- A text box permits text processing. Characters or words can wrap from line to line in the text box. You can give name and display attributes to the object.

- Programs can manipulate the object using the name. The name associates a variable text object with its variable definition. The initial value comes from the value specified for the object and is displayed using the output format defined for the variable and display attributes specified for the object.

- Objects can be line or box graphics, constant or variable text. They can occupy a single line or a rectangular area (box) on the form.

Creating a Stored Object

Purpose

FDP$CREATE_STORED_OBJECT procedure creates an initial value for a table variable occurrence that does not initially appear on the form.

Format

FDP$CREATE_STORED_OBJECT (form_identifier, name, occurrence, text, display_attribute_set, status)

Parameters

form_identifier: fdt$form_identifier;

The form identifier established when the form was opened.

name: ost$name;

The object name.

occurrence: fdt$occurrence;

The occurrence of the object.

text: fdt$text;

The text specifying the initial value.

display_attribute_set: fdt$display_attribute_set;

Specifies the set of display attributes for the object.

status: VAR of ost$status;

The status variable in which the completion status is returned.

Conditions

fde$bad_data_value
fde$cannot_update_opened_form
fde$invalid_form_identifier
fde$invalid_object_name
fde$invalid_occurrence
fde$no_space_available
fde$no_string_specified
fde$object_exists
fde$object_occurrence_exists
fde$system_error

Remarks

- The visible occurrences of a table initially appear on the form. The user can execute paging or scrolling events to look at the stored occurrences.

- If this procedure is not issued, the initial value for a stored object is the first occurrence of the object. A table can have one or more variables, and a variable in a table can have one or more occurrences.

Creating a Table

Purpose FDP$CREATE_TABLE procedure creates a table containing variables.

Format **FDP$CREATE_TABLE (form_identifier, table_name, table_attributes, status)**

Parameters form_identifier: fdt$form_identifier;

The form identifier established when the form was opened.

table_name: ost$name;

The table name.

table_attributes: VAR { input-output } of fdt$table_attributes;

The attributes of the table.

status: VAR of ost$status;

The status variable in which the completion status is returned.

Conditions
fde$cannot_update_opened_form
fde$invalid_form_identifier
fde$invalid_table_name
fde$no_space_available
fde$table_name_exists

Remarks

- The variables in a table can occur more than once and appear anywhere on the form.
- A table name cannot duplicate an existing table or variable name.
- You must create objects for table variable occurrences and variables for the table. When executing a FDP$END_FORM procedure, all variables and objects for the table must be defined.

Creating a Variable

Purpose FDP$CREATE_VARIABLE procedure creates a variable.

Format **FDP$CREATE_VARIABLE (form_identifier, variable_name, variable_attributes, status)**

Parameters form_identifier: fdt$form_identifier;

The form identifier established when the form was opened.

variable_name: ost$name;

The name of the variable.

variable_attributes: VAR { input-output } of fdt$variable_attributes;

An array containing variable attributes.

status: VAR of ost$status;

The status variable in which the completion status is returned.

Conditions

fde$bad_data_value
fde$cannot_update_opened_form
fde$invalid_form_identifier
fde$invalid_form_name
fde$invalid_integer_range
fde$invalid_real_range
fde$invalid_variable_name
fde$no_comments_to_delete
fde$no_space_available
fde$no_string_specified
fde$range_overlap
fde$system_error
fde$unknown_integer_range
fde$unknown_real_range
fde$unknown_valid_string
fde$valid_string_exists
fde$variable_name_exists

Remarks

- Every variable that appears to the user must have an object associated with it. The object can be created before or after the creation of the variable. Some variables are not shown on the form.
- Issue an FDP$CREATE_OBJECT procedure to specify the initial value and display attributes for a variable appearing on the form.

Deleting an Area

Purpose

FDP$DELETE_AREA procedure deletes all objects and unprotected text in a specified area on the form. Any associated table and variable definitions are not deleted.

Format

FDP$DELETE_AREA (form_identifier, x_position, y_position, width, height, status)

Parameters

form_identifier: fdt$form_identifier;

The form identifier established when the form was opened.

x_position: fdt$x_position;

The x position of the origin (upper left corner) of the area enclosing the data to be deleted.

y_position: fdt$y_position;

The y position of the origin (upper left corner) of the area enclosing the data to be deleted.

width: fdt$width;

The width of the area.

height: fdt$height;

The height of the area.

status: VAR of ost$status;

The status variable in which the completion status is returned.

Conditions

fde$area_cuts_object
fde$bad_data_value
fde$delete_outside_form
fde$invalid_form_identifier
fde$no_space_available
fde$system_error

Deleting a Mark

Purpose FDP$DELETE_MARK procedure deletes the previous mark set by the FDP$CREATE_MARK procedure.

Format **FDP$DELETE_MARK (form_identifier, status)**

Parameters form_identifier: fdt$form_identifier;

The form identifier established when the form was opened.

status: VAR of ost$status;

The status variable in which the completion status is returned.

Conditions
fde$bad_data_value
fde$delete_mark_invalid
fde$form_not_scheduled
fde$form_pushed
fde$invalid_form_identifier
fde$no_space_available
fde$system_error

Remarks This procedure can only be used on a form created with the FDP$CREATE_DESIGN_FORM procedure.

Deleting an Object

Purpose FDP$DELETE_OBJECT procedure deletes an object at a specified location on the form. Any variable or table definitions associated with the object are not deleted.

Format **FDP$DELETE_OBJECT (form_identifier, x_position, y_position, status)**

Parameters form_identifier: fdt$form_identifier;

The form identifier established when the form was opened.

x_position: fdt$x_position;

The x position of the object to delete.

y_position: fdt$y_position;

The y position of the object to delete.

status: VAR of ost$status;

The status variable in which the completion status is returned.

Conditions fde$cannot_update_opened_form
fde$invalid_form_identifier
fde$no_object_at_position

Deleting a Stored Object

Purpose FDP$DELETE_STORED_OBJECT procedure deletes an initial value for a table variable occurrence that does not initially appear on a form.

Format **FDP$DELETE_STORED_OBJECT (form_identifier, name, occurrence, status)**

Parameters form_identifier: fdt$form_identifier;

The form identifier established when the form was opened.

name: ost$name;

The object name.

occurrence: fdt$occurrence;

The occurrence of the object.

status: VAR of ost$status;

The status variable in which the completion status is returned.

Conditions

fde$cannot_update_opened_form
fde$invalid_form_identifier
fde$invalid_object_name
fde$no_space_available
fde$system_error
fde$unknown_object_name

Deleting a Table

Purpose FDP$DELETE_TABLE procedure deletes a table. Any variables or object definitions associated with the table are not deleted.

Format **FDP$DELETE_TABLE (form_identifier, table_name, status)**

Parameters form_identifier: fdt$form_identifier;

The form identifier established when the form was opened.

table_name: ost$name;

The table to be deleted.

status: VAR of ost$status;

The status variable in which the completion status is returned.

Conditions

fde$bad_data_value
fde$cannot_update_opened_form
fde$invalid_form_identifier
fde$invalid_table_name
fde$unknown_table_name

Deleting a Variable

Purpose

FDP$DELETE_VARIABLE procedure deletes a variable. Any table or object definitions associated with the variable are not deleted.

Format

FDP$DELETE_VARIABLE (form_identifier, variable_name, status)

Parameters

form_identifier: integer;

The form identifier established when the form was opened.

variable_name: name;

The variable to be deleted.

status: VAR of status;

The status variable in which the completion status is returned.

Conditions

fde$bad_data_value
fde$cannot_update_opened_form
fde$invalid_form_identifier
fde$invalid_variable_name
fde$unknown_variable_name

Editing a Form

Purpose FDP$EDIT_FORM procedure permits you to make further changes to a copied form or a previously ended form definition.

Format **FDP$EDIT_FORM (form_identifier, status)**

Parameters form_identifier: fdt$form_identifier;

The form identifier established when the form was opened.

status: VAR of ost$status;

The status variable in which the completion status is returned.

Conditions fde$bad_data_value
fde$cannot_update_opened_form
fde$invalid_form_identifier

Ending a Form

Purpose FDP$END_FORM procedure ends the definition of a form.

Format **FDP$END_FORM (form_identifier, p_sequence, number_errors, p_errors, status)**

Parameters form_identifier: fdt$form_identifier;

The form identifier established when the form was opened.

p_sequence: ^SEQ(*);

The sequence to return any errors.

number_errors: VAR of fdt$number_errors;

The number of errors contained in the form definition.

p_errors: VAR of ^SEQ (*);

The sequence that contains the errors.

status: VAR of ost$status;

The status variable in which the completion status is returned.

Conditions
fde$bad_data_value
fde$cannot_update_opened_form
fde$invalid_form_identifier
fde$no_space_available
fde$system_error

Remarks This procedure must be executed before you can use a form to interact with a terminal user.

Getting Form Attributes

Purpose FDP$GET_FORM_ATTRIBUTES procedure gets the current form attributes. The form must be open or dynamically created.

Format **FDP$GET_FORM_ATTRIBUTES (form_identifier, get_form_attributes, status)**

Parameters form_identifier: fdt$form_identifier;

The form identifier established when the form was opened.

get_form_attributes: VAR { input-output } of fdt$get_form_attributes;

An array containing form attributes.

status: VAR of ost$status;

The status variable in which the completion status is returned.

Conditions
fde$bad_data_value
fde$cannot_update_opened_form
fde$invalid_event_name
fde$invalid_form_identifier
fde$string_too_small
fde$system_error
fde$unknown_event_name

Getting Form Names

Purpose

FDP$GET_FORM_NAMES procedure gets the current names of tables, variables, and objects defined for a form.

Format

FDP$GET_FORM_NAMES (form_identifier, name_selections, form_names, number_names, status)

Parameters

form_identifier: fdt$form_identifier;

The form identifier established when the form was opened.

name_selections: fdt$name_selections;

A set containing selections for names. You can select FDC$SELECT_VARIABLE, FDC$SELECT_TABLE, and FDC$SELECT_OBJECT.

form_names: VAR of fdt$form_names;

An array containing the form names. The form names are contained in a record with name and name_type fields.

Field	Meaning
name	The item name (variable, table, object).
name_type	The name type (FDC$SELECT_VARIABLE, FDC$SELECT_TABLE, FDC$SELECT_OBJECT).

number_names: VAR of fdt$number_names;

The number of names returned.

status: VAR of ost$status;

The status variable in which the completion status is returned.

Conditions

fde$cannot_update_opened_form
fde$invalid_form_identifier
fde$too_many_form_names

Remarks

- Issue the FDP$GET_FORM_ATTRIBUTES procedure using the attribute key of FDC$GET_NUMBER_OBJECTS, FDC$GET_NUMBER_TABLES, and FDC$GET_NUMBER_VARIABLES.
- This procedure enables you to learn the array size needed to receive the form names.

Getting Form Objects

Purpose FDP$GET_FORM_OBJECTS procedure gets objects defined for a form.

Format **FDP$GET_FORM_OBJECTS (form_identifier, form_objects, number_objects, status)**

Parameters **form_identifier**: fdt$form_identifier;

The form identifier established when the form was opened.

form_objects: VAR of fdt$form_objects;

An array containing the form objects that Screen Formatting returns. Each record in the array has a name and an object field.

Field	Meaning
name	The object name. If the object did not have a name defined, the name equals OSC$NULL_NAME.
object	The object type. FDC$BOX The object is a box. FDC$CONSTANT_TEXT The object is constant text. FDC$CONSTANT_TEXT_BOX The object is a constant text box. FDC$LINE The object is a line. FDC$VARIABLE_TEXT The object is variable text. FDC$VARIABLE_TEXT_BOX The object is a variable text box.

Field	Meaning
occurrence	The occurrence of the object name. If the name is OSC$NULL_NAME, the occurrence equals 1.
x_position	The form x position of the object.
y_position	The form y position of the object.

number_objects: VAR of fdt$number_objects;

The number of objects returned from Screen Formatting.

status: VAR of ost$status;

The status variable in which the completion status is returned.

Conditions

fde$bad_data_value
fde$cannot_update_opened_form
fde$invalid_form_identifier
fde$too_many_form_objects

Remarks

Issue the FDP$GET_FORM_ATTRIBUTES procedure by using the attribute key of FDC$GET_NUMBER_OBJECTS. This procedure enables you to learn the array size needed to receive the form objects.

Getting Object Attributes

Purpose

FDP$GET_OBJECT_ATTRIBUTES procedure gets specified attributes about an object on the form. This procedure can be used on an open or dynamically created form.

Format

FDP$GET_OBJECT_ATTRIBUTES (form_identifier, x_position, y_position, get_object_attributes, status)

Parameters

form_identifier: fdt$form_identifier;

The form identifier established when the form was opened.

x_position: fdt$x_position;

The x position on the form.

y_position: fdt$y_position;

The y position on the form.

object_attributes: VAR { input-output } of fdt$get_object_attributes;

An array containing object attributes. Before you specify this parameter, you must first establish the array.

status: VAR of ost$status;

The status variable in which the completion status is returned.

Conditions

fde$bad_data_value
fde$cannot_update_opened_form
fde$invalid_form_identifier
fde$no_object_at_position
fde$system_error

Getting Record Attributes

Purpose FDP$GET_RECORD_ATTRIBUTES procedure gets form definition record attributes. You can execute this procedure on an open or dynamically created form.

Format **FDP$GET_RECORD_ATTRIBUTES (form_identifier, get_record_attributes, status)**

Parameters form_identifier: fdt$form_identifier;

The form identifier established when the form was opened.

get_record_attributes: VAR { input-output } of fdt$get_record_attributes;

An array containing form definition record attributes. Before you specify this parameter, you must first establish the array.

status: VAR of ost$status;

The status variable in which the completion status is returned.

Conditions fde$cannot_update_opened_form
fde$invalid_form_identifier
fde$invalid_table_name
fde$invalid_variable_name
fde$system_error
fde$unknown_occurrence
fde$unknown_table_name

Getting a Stored Object

Purpose FDP$GET_STORED_OBJECT procedure gets the initial value for a table variable occurrence that does not appear initially on a form.

Format **FDP$GET_STORED_OBJECT (form_identifier, name, occurrence, text, text_length, status)**

Parameters form_identifier: fdt$form_identifier;

The form identifier established when the form was opened.

name: ost$name;

The object name.

occurrence: fdt$occurrence;

The occurrence of the object.

text: VAR of fdt$text;

The text specifying the initial value.

text_length: VAR of fdt$text_length;

The stored object length. This length can exceed the parameter text length. Allocate more space for the text and re-issue the procedure if the text_length is greater than the allocated space for the text.

status: VAR of ost$status;

The status variable in which the completion status is returned.

Conditions

fde$invalid_address
fde$invalid_form_identifier
fde$invalid_object_name
fde$invalid_occurrence
fde$no_space_available
fde$no_string_specified
fde$system_error
fde$system_error
fde$unknown_object_name

Getting Table Attributes

Purpose FDP$GET_TABLE_ATTRIBUTES gets procedure-specified table attributes. You can execute this procedure on an open or dynamically created form.

Format **FDP$GET_TABLE_ATTRIBUTES (form_identifier, get_table_attributes, status)**

Parameters form_identifier: fdt$form_identifier;

The form identifier established when the form was opened.

table_name: ost$name;

The table name.

get_table_attributes: VAR { input-output } of fdt$get_table_attributes;

Table attributes.

status: VAR of ost$status;

The status variable in which the completion status is returned.

Conditions fde$bad_data_vaule
fde$invalid_form_identifier
fde$no_object_at_position
fde$system_error

Getting Variable Attributes

Purpose FDP$GET_VARIABLE_ATTRIBUTES procedure gets selected information about a variable. You can execute this procedure on an open or dynamically created form.

Format **FDP$GET_VARIABLE_ATTRIBUTES (form_identifier, variable name, get_variable_attributes, status)**

Parameters form_identifier: fdt$form_identifier;

The form identifier established when the form was opened.

variable_name: ost$name;

The variable name.

get_variable_attributes: VAR { input-output } of fdt$get_variable_attributes;

Gets an array containing attributes.

status: VAR of ost$status;

The status variable in which the completion status is returned.

Conditions

fde$bad_data_vaule
fde$cannot_update_opened_form
fde$invalid_form_identifier
fde$invalid_variable_name
fde$string_too_small
fde$system_error
fde$unknown_variable_name

Moving an Area

Purpose

FDP$MOVE_AREA procedure moves all objects and unprotected text from one area of a form to another. The destination area cannot slice any objects outside the origin area.

Format

FDP$MOVE_AREA (form_identifier, from_x_position, from_y_position, width, height, to_x_position, to_y_position, status)

Parameters

form_identifier: fdt$form_identifier;

The form identifier established when the form was opened.

from_x_position: fdt$x_position;

The x position of the origin (upper left_hand corner) of the area enclosing the data to be moved.

from_y_position: fdt$y_position;

The y position of the origin (upper left_hand corner) of the area enclosing the data to be moved.

width: fdt$width;

The width of the area.

height: fdt$height;

The height of the area.

to_x_position: fdt$x_position;

The x position of the area (upper left_hand corner) where the data is to be copied.

to_y_position: fdt$y_position;

The y position of the area (upper left_hand corner) where the data is to be copied.

status: VAR of ost$status

The status variable in which the completion status is returned.

Conditions

fde$bad_data_value
fde$invalid_form_identifier
fde$move_outside_form
fde$no_space_available
fde$object_overlays
fde$system_error

Writing a Form Definition

Purpose FDP$WRITE_FORM_DEFINITION procedure writes a form to a segment access file.

Format **FDP$WRITE_FORM_DEFINITION (form_identifier, p_form_module, status)**

Parameters form_identifier: fdt$form_identifier;

The form identifier established when the form was opened.

p_form_module: VAR { input-output } of ^SEQ (*);

A pointer to a sequence that holds the form module.

status: VAR of ost$status;

The status variable in which the completion status is returned.

Conditions fde$bad_data_value
fde$invalid_form_identifier
fde$no_space_available

Remarks A segment access file can be used with the CREATE_OBJECT_LIBRARY command to save a form. The form does not have to be ended and can contain errors. The form must have a non-blank name.

Writing a Form Definition Record

Purpose FDP$WRITE_RECORD_DEFINITION procedure writes the Source Code Utility (SCU) deck defining the record to transfer data between the program and Screen Formatting.

Format **FDP$WRITE_RECORD_DEFINITION (form_identifier, file_identifier, form_processor, status)**

Parameters form_identifier: fdt$form_identifier;

The form identifier established when the form was opened.

file_identifier: amt$file_identifier;

The file identifier returned by the FSP$OPEN_FILE request that opened the file. This is the file to which the deck is written.

form_processor: fdt$form_processor;

The processor that uses the record definition.

status: VAR of ost$status;

The status variable in which the completion status is returned.

Conditions

fde$bad_data_value
fde$form_definition_errors
fde$form_has_no_variables
fde$form_not_ended
fde$invalid_form_identifier
fde$invalid_form_processor

Remarks

- The form cannot have any errors and must have ended with the FDP$END_FORM procedure.
- The resulting file contains an SCU DECK directive that, when processed, gives the deck the same name as the form. You can change this name by using the FDP$CHANGE_FORM_RECORD procedure. Include a deck name in the record attributes.

Glossary A

A

Alphabetic Character

One of the following letters:

A through Z
a through z

See also Character.

Attribute

A property of a form, variable, table of variables, object, or constant that is needed to process a form.

B

Batch Mode

A mode of execution in which a job is submitted and processed as a unit with no intervention from a user. Contrast with Interactive Mode.

C

Catalog

A directory of files maintained by the operating system for a user. In addition to files, a catalog can contain other catalogs. The catalog $LOCAL contains only file entries.

Catalog Name

The name of a catalog in a catalog hierarchy. By convention, the name of the user's master catalog is the same as the user's user name.

Character

An alphabetic character, digit, space, or symbol. See also Alphabetic Character, Digit, and Symbol.

D

Design Form

One of two types of forms necessary for enabling a terminal user to interact with an application program in order to create, display, or change a form.

Digit

One of the following characters:

0 1 2 3 4 5 6 7 8 9

See also Character.

E

Event

A property of a form that is defined by the application programmer. An example would be a function key.

F

Family

A logical grouping of NOS/VE users that determines the location of their permanent files.

Family Name

A name that identifies a NOS/VE family.

File

A collection of records referenced by a file name. A file is an autonomous collection of information that exists separately from the programs that read or write the file.

File Name

The name of a NOS/VE file. See also Name.

Full Screen

A program that utilizes the entire terminal screen to display the data and/or the user's options. The user can move the cursor around on the screen to modify data or to indicate which operation to execute.

Full Screen Definition

Instructions to NOS/VE describing the full screen features and function keys for a terminal. To run full screen programs on a terminal, NOS/VE needs a full screen definition for the type of terminal used.

Function

An instruction to a full screen program. If function keys are available on the keyboard, the user can press a function key to execute a function.

Function Key

A key on a keyboard that is used to execute a function. Function keys are often labelled with an F and a digit. For example, F1, F2, or F3.

Function Key Assignments

The association of functions with the function keys on the user's keyboard. In most full screen programs, the function key assignments are displayed at the bottom of the screen.

I

Identifier

A character of group of characters that identify items of data.

Integer

Numeric data (positive or negative) that represents a whole number. An integer is stored internally as a binary value rather than as a character value.

Interactive Mode

A mode of execution during which a user enters commands, subcommands, or functions at a terminal and the computer responds immediately to each command, subcommand, or function.

L

Local File

A file that is accessed via the $LOCAL catalog as follows:

    $LOCAL.filename

NOS/VE discards all $LOCAL files when the user logs out. Contrast with Permanent File.

Login

The process used at a terminal to gain access to an operating system such as NOS/VE. Logging in starts a terminal session.

Logout

The process used at a terminal to end a terminal session.

M

Main Menu

The menu that is available at the beginning of a program or online manual.

Master Catalog

The catalog the operating system creates for each user name. The user's master catalog contains entries for all permanent files and catalogs a user creates. By convention, the name of the master catalog is the same as the user name.

N

Name

A combination of 1 through 31 characters chosen from the following:

Alphabetic characters (A through Z and a through z)
Digits (0 through 9)
Special characters (#, @, $, or _)

The first character of a name cannot be a digit.

NOS/VE

Network Operating System/Virtual Environment.

O

Object

An object can be constant or variable text, box drawing, line drawing, or a table that contains one or more occurrences of one or more variable text objects.

Occurrence

The number of times an object appears on a form.

Online Manual

A manual that the user reads on the terminal screen. The EXPLAIN command opens an online manual.

P

Permanent Catalog

A catalog of permanent files, such as the master catalog or a catalog within the master catalog.

Permanent File

A file that is accessed via the user's master catalog. Permanent files are not discarded when the user logs out of NOS/VE. Contrast with Local File.

Program

A set of instructions or actions that can interface with Screen Formatting.

Protected and Unprotected Text

Protected text is text that cannot be changed by the terminal user. Unprotected text can be changed by the terminal user.

S

SCL

See System Command Language

Special Character

See Symbol.

Symbol

Any character that is not an alphabetic character or a digit. Examples are: #, $, %, &, and *. See also Character.

System Command Language

The language that provides the interface to the features and capabilities of NOS/VE.

T

Target Form

The desired form that is created from the design form.

Temporary File

See Local File.

Terminal Session

The processing sequence that begins when a user logs in to an operating system and ends when the user logs out.

U

User Name

A name that identifies a NOS/VE user.

Related Manuals B

This appendix lists the manuals which relate to NOS/VE. Also included is information for ordering printed manuals and the way to access online manuals.

Ordering Printed Manuals B-1

Accessing Online Manuals B-1

Table B-1. Related Manuals B-2
NOS/VE Site Manuals B-2
NOS/VE User Manuals B-3
CYBIL Manuals B-5
FORTRAN Manuals B-6
COBOL Manuals B-6
Other Compiler Manuals B-7
VX/VE Manuals B-8
Data Management Manuals B-10
Information Management Manuals B-10
CDCNET Manuals B-11
Migration Manuals B-13
Miscellaneous Manuals B-13
Hardware Manuals B-15

Related Manuals B

All NOS/VE manuals and related hardware manuals are listed in table B-1. If your site has installed the online manuals, you can find an abstract for each NOS/VE manual in the online System Information manual. To access this manual, enter:

```
/help manual=nos_ve
```

Ordering Printed Manuals

To order a printed Control Data manual, send an order form to:

Control Data
Literature and Distribution Services
308 North Dale Street
St. Paul, Minnesota 55103-2495

To obtain an order form or to get more information about ordering Control Data manuals, write to the above address or call (612) 292-2101. If you are a Control Data employee, call (612) 292-2100.

Accessing Online Manuals

To access the online version of a printed manual, log in to NOS/VE and enter the online title on the HELP command (table B-1 supplies the online titles). For example, to see the NOS/VE Commands and Functions manual, enter:

```
/help manual=scl
```

or, because SCL is the default for the MANUAL parameter, simply enter

```
/help
```

An online Examples manual contains examples that reside in printed manuals. From within the online Examples manual, you can copy, print, and execute the examples it contains. To access this manual, enter:

```
/help manual=examples
```

When EXAMPLES is listed in the Online Manuals column in table B-1, that manual is represented in the online Examples manual.

Table B-1. Related Manuals

Manual Title	Publication Number	Online Manuals[1]
NOS/VE Site Manuals:		
CYBER 930 Computer System Guide to Operations Usage	60469560	
CYBER Initialization Package (CIP) Reference Manual	60457180	
Desktop/VE Host Utilities Usage	60463918	
MAINTAIN_MAIL (Version 1) [2] Usage		MAIM
NOS/VE Accounting Analysis System Usage	60463923	
NOS/VE Accounting and Validation Utilities for Dual State Usage	60458910	
NOS/VE File Server for STORNET and ESM-II Usage	60000190	
NOS/VE LCN Configuration and Network Management Usage	60463917	

1. This column lists the title of the online version of the manual and indicates whether the examples in the printed manual are in the online Examples manual.

2. To access this manual, you must be the administrator for Mail/VE Version 1.

(Continued)

Table B-1. Related Manuals *(Continued)*

Manual Title	Publication Number	Online Manuals[1]
Site Manuals (Continued):		
NOS/VE Network Management Usage	60463916	
NOS/VE Operations Usage	60463914	
NOS/VE System Performance and Maintenance Volume 1: Performance Usage	60463915	
NOS/VE System Performance and Maintenance Volume 2: Maintenance Usage	60463925	
NOS/VE User Validation Usage	60464513	
NOS/VE User Manuals:		
EDIT_CATALOG Usage		EDIT_ CATALOG
EDIT_CATALOG for NOS/VE Summary	60487719	
Introduction to NOS/VE Tutorial	60464012	EXAMPLES
NOS/VE Advanced File Management Tutorial	60486412	AFM_T

1. This column lists the title of the online version of the manual and indicates whether the examples in the printed manual are in the online Examples manual.

(Continued)

Table B-1. Related Manuals *(Continued)*

Manual Title	Publication Number	Online Manuals[1]
NOS/VE User Manuals (Continued):		
NOS/VE Advanced File Management Usage	60486413	AFM
NOS/VE Advanced File Management Summary	60486419	
NOS/VE Commands and Functions Quick Reference	60464018	SCL
NOS/VE File Editor Tutorial/Usage	60464015	EXAMPLES
NOS/VE Object Code Management Usage	60464413	OCM and EXAMPLES
NOS/VE Screen Formatting Usage	60488813	EXAMPLES
NOS/VE Source Code Management Usage	60464313	SCM and EXAMPLES
NOS/VE System Usage	60464014	EXAMPLES
NOS/VE Terminal Definition Usage	60464016	
Screen Design Facility for NOS/VE Usage	60488613	SDF

1. This column lists the title of the online version of the manual and indicates whether the examples in the printed manual are in the online Examples manual.

(Continued)

Table B-1. Related Manuals *(Continued)*

Manual Title	Publication Number	Online Manuals[1]
NOS/VE User Manuals (Continued):		
Screen Design Facility/Data Management Usage	60488618	
CYBIL Manuals:		
CYBIL for NOS/VE File Management Usage	60464114	EXAMPLES
CYBIL for NOS/VE Keyed-File and Sort/Merge Interfaces Usage	60464117	EXAMPLES
CYBIL for NOS/VE Language Definition Usage	60464113	CYBIL and EXAMPLES
CYBIL for NOS/VE Sequential and Byte-Addressable Files Usage	60464116	EXAMPLES
CYBIL for NOS/VE System Interface Usage	60464115	EXAMPLES

1. This column lists the title of the online version of the manual and indicates whether the examples in the printed manual are in the online Examples manual.

(Continued)

Table B-1. Related Manuals *(Continued)*

Manual Title	Publication Number	Online Manuals[1]
FORTRAN Manuals:		
FORTRAN Version 1 for NOS/VE Language Definition Usage	60485913	EXAMPLES
FORTRAN Version 1 for NOS/VE Quick Reference		FORTRAN
FORTRAN Version 2 for NOS/VE Language Definition Usage	60487113	EXAMPLES
FORTRAN Version 2 for NOS/VE Quick Reference		VFORTRAN
FORTRAN for NOS/VE Tutorial	60485912	FORTRAN_T
FORTRAN for NOS/VE Topics for FORTRAN Programmers Usage	60485916	
FORTRAN for NOS/VE Summary	60485919	
COBOL Manuals:		
COBOL for NOS/VE Summary	60486019	

1. This column lists the title of the online version of the manual and indicates whether the examples in the printed manual are in the online Examples manual.

(Continued)

Table B-1. Related Manuals *(Continued)*

Manual Title	Publication Number	Online Manuals[1]
COBOL Manuals (Continued):		
COBOL for NOS/VE Tutorial	60486012	COBOL_T
COBOL for NOS/VE Usage	60486013	COBOL and EXAMPLES
Other Compiler Manuals:		
ADA for NOS/VE Usage	60498113	ADA
ADA for NOS/VE Reference Manual	60498118	EXAMPLES
APL for NOS/VE File Utilities Usage	60485814	
APL for NOS/VE Language Definition Usage	60485813	
BASIC for NOS/VE Summary Card	60486319	
BASIC for NOS/VE Usage	60486313	BASIC
LISP for NOS/VE Usage Supplement	60486213	
Pascal for NOS/VE Summary Card	60485619	

1. This column lists the title of the online version of the manual and indicates whether the examples in the printed manual are in the online Examples manual.

(Continued)

Table B-1. Related Manuals *(Continued)*

Manual Title	Publication Number	Online Manuals[1]
Other Compiler Manuals (Continued):		
Pascal for NOS/VE Usage	60485618	PASCAL and EXAMPLES
Prolog for NOS/VE Quick Reference	60486718	PROLOG
Prolog for NOS/VE Usage	60486713	
VX/VE Manuals:		
C/VE for NOS/VE Quick Reference		C
C/VE for NOS/VE Usage	60469830	
DWB/VX Introduction and User Reference Tutorial/Usage	60469890	
DWB/VX Macro Packages Guide Usage	60469910	
DWB/VX Preprocessors Guide Usage	60469920	
DWB/VX Text Formatters Guide Usage	60469900	

1. This column lists the title of the online version of the manual and indicates whether the examples in the printed manual are in the online Examples manual.

(Continued)

Table B-1. Related Manuals *(Continued)*

Manual Title	Publication Number	Online Manuals[1]
VX/VE Manuals (Continued):		
VX/VE Administrator Guide and Reference Tutorial/Usage	60469770	
VX/VE An Introduction for UNIX Users Tutorial/Usage	60469980	
VX/VE Programmer Guide Tutorial	60469790	
VX/VE Programmer Reference Usage	60469820	
VX/VE Support Tools Guide Tutorial	60469800	
VX/VE User Guide Tutorial	60469780	
VX/VE User Reference Usage	60469810	

1. This column lists the title of the online version of the manual and indicates whether the examples in the printed manual are in the online Examples manual.

(Continued)

Table B-1. Related Manuals *(Continued)*

Manual Title	Publication Number	Online Manuals[1]
Data Management Manuals:		
DM Command Procedures Reference Manual	60487905	
DM Concepts and Facilities Manual	60487900	
DM Error Message Summary for DM on CDC NOS/VE	60487906	
DM Fundamental Query and Manipulation Manual	60487903	
DM Report Writer Reference Manual	60487904	
DM System Administrator's Reference Manual for DM on CDC NOS/VE	60487902	
DM Utilities Reference Manual for DM on CDC NOS/VE	60487901	
Information Management Manuals:		
IM/Control for NOS/VE Quick Reference	L60488918	CONTROL
IM/Control for NOS/VE Usage	60488913	
IM/Fast for NOS/VE Administration Usage	60487513	

1. This column lists the title of the online version of the manual and indicates whether the examples in the printed manual are in the online Examples manual.

(Continued)

Table B-1. Related Manuals *(Continued)*

Manual Title	Publication Number	Online Manuals[1]
Information Management Manuals (Continued):		
IM/Fast for NOS/VE Programming Usage	60487514	
IM/Quick for NOS/VE Tutorial	60485712	
IM/Quick for NOS/VE Summary	60485714	
IM/Quick for NOS/VE Online Help		QUICK
CDCNET Manuals:		
CDCNET Access Guide		CDCNET_ ACCESS
CDCNET Batch Device User Guide		CDCNET_ BATCH
CDCNET Command Quick Reference	60000020	
CDCNET Conceptual Overview	60461540	
CDCNET Configuration and Site Administration Guide	60461550	
CDCNET DI Dump Analyzer		ANACD
CDCNET Diagnostic Messages	60461600	CDCNET_ MSGS

1. This column lists the title of the online version of the manual and indicates whether the examples in the printed manual are in the online Examples manual.

(Continued)

Table B-1. Related Manuals *(Continued)*

Manual Title	Publication Number	Online Manuals[1]
CDCNET Manuals (Continued):		
CDCNET Network Analysis	60461590	
CDCNET Network Configuration Utility		NETCU
CDCNET Network Configuration Utility Summary Card	60000269	
CDCNET Network Operations	60461520	
CDCNET Network Performance Analyzer	60461510	
CDCNET Product Descriptions	60460590	
CDCNET Systems Programmer's Reference Manual Volume 1 Base System Software	60462410	
CDCNET Systems Programmer's Reference Manual Volume 2 Network Management Entities and Layer Interfaces	60462420	
CDCNET Systems Programmer's Reference Manual Volume 3 Network Protocols	60462430	
CDCNET Terminal Interface Usage	60463850	
CDCNET TCP/IP Usage	60000214	

1. This column lists the title of the online version of the manual and indicates whether the examples in the printed manual are in the online Examples manual.

(Continued)

Table B-1. Related Manuals *(Continued)*

Manual Title	Publication Number	Online Manuals[1]
Migration Manuals:		
Migration from IBM to NOS/VE Tutorial/Usage	60489507	
Migration from NOS to NOS/VE Tutorial/Usage	60489503	
Migration from NOS to NOS/VE Standalone Tutorial/Usage	60489504	
Migration from NOS/BE to NOS/VE Tutorial/Usage	60489505	
Migration from NOS/BE to NOS/VE Standalone Tutorial/Usage	60489506	
Migration from VAX/VMS to NOS/VE Tutorial/Usage	60489508	
Miscellaneous Manuals:		
Applications Directory	60455370	
Control Data CONNECT User's Guide	60462560	
Control Data CONNECT Plus for the IBM Personal Computer (Version 1.0) User's Guide	60000388	

1. This column lists the title of the online version of the manual and indicates whether the examples in the printed manual are in the online Examples manual.

(Continued)

Table B-1. Related Manuals *(Continued)*

Manual Title	Publication Number	Online Manuals[1]
Miscellaneous Manuals (Continued):		
Debug for NOS/VE Quick Reference		DEBUG
Debug for NOS/VE Usage	60488213	
Desktop/VE for Macintosh Tutorial	60464502	
Desktop/VE for Macintosh Usage	60464503	
NOS/VE Diagnostic Messages Usage	60464613	MESSAGES
Mail/VE (Version 1) Summary Card	60464519	
Mail/VE (Version 1) Usage		MAIL_VE
Mail/VE Version 2 Administration	60464515	MAILVE_ ADMINISTRA- TION
Mail/VE Version 2 Usage	60464514	MAILVE_V2
Math Library for NOS/VE Usage	60486513	
NOS/VE Examples Usage		EXAMPLES

1. This column lists the title of the online version of the manual and indicates whether the examples in the printed manual are in the online Examples manual.

(Continued)

Table B-1. Related Manuals *(Continued)*

Manual Title	Publication Number	Online Manuals[1]
Miscellaneous Manuals (Continued):		
NOS/VE Online Manuals Systems	60488403	TOPICS_ CONTEXT
NOS/VE System Information		NOS_VE
Programming Environment for NOS/VE Usage		ENVIRON-MENT
Programming Environment for NOS/VE Summary	60486819	
Professional Programming Environment for NOS/VE Quick Reference		PPE
Professional Programming Environment for NOS/VE Usage	60486613	
Remote Host Facility Usage	60460620	
Hardware Manuals:		
CYBER 170 Computer Systems Models 825, 835, and 855 General Description Hardware Reference	60459960	

1. This column lists the title of the online version of the manual and indicates whether the examples in the printed manual are in the online Examples manual.

(Continued)

Table B-1. Related Manuals *(Continued)*

Manual Title	Publication Number	Online Manuals[1]
Hardware Manuals (Continued):		
CYBER 170 Computer Systems, Models 815, 825, 835, 845, and 855 CYBER 180 Models 810, 830, 835, 840, 845, 850, 855, and 860 Codes Booklet	60458100	
CYBER 170 Computer Systems, Models 815, 825, 835, 845, and 855 CYBER 180 Models 810, 830, 835, 840, 845, 850, 855, and 860 Maintenance Register Codes Booklet	60458110	
HPA/VE Reference	60461930	
Virtual State Volume II Hardware Reference	60458890	
7021-31/32 Advanced Tape Subsystem Reference	60449600	
7221-1 Intelligent Small Magnetic Tape Subsystem Reference	60461090	

1. This column lists the title of the online version of the manual and indicates whether the examples in the printed manual are in the online Examples manual.

Screen Formatting and Terminal Definitions

C

Screen Formatting and Terminal Definitions C

Here is a list of Screen Formatting and terminal definition attributes. Screen Formatting attributes are mapped to terminal definition attributes. Changing a terminal definition attribute can change how a Screen Formatting attribute is displayed on the screen.

Screen Formatting Attribute	Terminal Definition Attribute
fdc$inverse_video	inverse_begin
fdc$low_intensity	low_intensity_begin
fdc$high_intensity	high_intensity_begin
fdc$blink	blink_begin
fdc$underline	underline_begin
fdc$protect	protect_begin
fdc$black_foreground	black_foreground
fdc$blue_foreground	blue_foreground
fdc$green_foreground	green_foreground
fdc$magenta_foreground	magenta_foreground
fdc$red_foreground	red_foreground
fdc$cyan_foreground	cyan_foreground
fdc$yellow_foreground	yellow_foreground
fdc$white_foreground	white_foreground
fdc$black_background	black_background
fdc$blue_background	blue_background
fdc$green_background	green_background
fdc$magenta_background	magenta_background
fdc$red_background	red_background
fdc$cyan_background	cyan_background
fdc$yellow_background	yellow_background
fdc$white_background	white_background
fdc$fine_line	ld_fine_begin
fdc$medium_line	ld_medium_begin
fdc$bold_line	ld_bold_begin
fdc$fine_border	ld_fine_begin
fdc$medium_border	ld_medium_begin
fdc$bold_border	ld_bold_begin
fdc$italic_display_attribute	italic_begin
fdc$title_display_attribute	title_begin
fdc$input_display_attribute	input_text_begin
fdc$error_display_attribute	error_begin
fdc$message_display_attribute	message_begin

Screen Formatting uses for defaults:

fdc$black_background, fdc$white_foreground for forms

fdc$medium_line for lines and boxes

fdc$inverse_video for event label text in event forms

fdc$underline for design attributes of objects that do not have any other display attributes

Here is a list of Screen Formatting event triggers and the appropriate terminal definition keys. The Screen Formatting event trigger maps the Screen Formatting definitions to the terminal definitions.

Screen Formatting Event Trigger	Terminal Definition Keys
fdc$next	next
fdc$shift_next	next_s
fdc$help	help
fdc$shift_help	help_s
fdc$stop	stop
fdc$shift_stop	stop_s
fdc$back	back
fdc$undo	undo
fdc$redo	redo
fdc$quit	stop_s
fdc$exit	stop
fdc$shift_back	back_s
fdc$up	up
fdc$shift_up	up_s
fdc$down	down
fdc$shift_down	down_s
fdc$foreward	fwd
fdc$shift_foreward	fwd_s
fdc$backward	bkw
fdc$shift_backward	bkw_s
fdc$edit	edit
fdc$shift_edit	edit_s
fdc$data	data
fdc$shift_data	data_s
fdc$function_1	f1
fdc$shift_function_1	f1_s

Screen Formatting Event Trigger	Terminal Definition Keys
fdc$function_2	f2
fdc$shift_function_2	f2_s
fdc$function_3	f3
fdc$shift_function_3	f3_s
fdc$function_4	f4
fdc$shift_function_4	f4_s
fdc$function_5	f5
fdc$shift_function_5	f5_s
fdc$function_6	f6
fdc$shift_function_6	f6_s
fdc$function_7	f7
fdc$shift_function_7	f7_s
fdc$function_8	f8
fdc$shift_function_8	f8_s
fdc$function_9	f9
fdc$shift_function_9	f9_s
fdc$function_10	f10
fdc$shift_function_10	f10_s
fdc$function_11	f11
fdc$shift_function_11	f11_s
fdc$function_12	f12
fdc$shift_function_12	f12_s
fdc$function_13	f13
fdc$shift_function_13	f13_s
fdc$function_14	f14
fdc$shift_function_14	f14_s
fdc$function_15	f15
fdc$shift_function_15	f15_s
fdc$function_16	f16
fdc$shift_function_16	f16_s

COBOL Parameter Definitions D

FDE$COBOL_STATUS Deck D-1
FDE$COBOL_VARIABLE_STATUS Deck D-6

COBOL Parameter Definitions D

This appendix contains the COBOL parameter definitions. Your COBOL program should copy the FDE$COBOL_STATUS deck into the program to obtain the conditions for the COBOL status parameter (see chapter 3, Using COBOL to Manage Forms). Your program should also copy the FDE$COBOL_VARIABLE_STATUS deck into the program to obtain the conditions for the COBOL variable status parameter. Errors are then generated (if they occur) when the program is run.

See chapter 3 for details on how to obtain the decks that contain the COBOL parameter definitions. The library,

$SYSTEM.CYBIL.OSF$PROGRAM_INTERFACE

contains the information you need to execute the COBOL program.

FDE$COBOL_STATUS Deck

The contents of this deck follow.

```
01  FDE-COBOL-STATUS USAGE COMP PIC S9(18) SYNC LEFT.
    88 FDE-REQUEST-SUCCESSFUL VALUE 0.
    88 FDE-TERMINAL-DISCONNECTED VALUE 1.
    88 FDE-NO-INPUT-REQUEST VALUE 2.
    88 FDE-CURSOR-NOT-IN-VARIABLE VALUE 3.

    88 FDE-MORE-ERRORS-EXIST VALUE 4.
    88 FDE-UNKNOWN-FORM-NAME VALUE 5.
    88 FDE-FORM-COMPILATION-ERRORS VALUE 6.
    88 FDE-NO-SPACE-AVAILABLE VALUE 7.

    88 FDE-UNSUPPORTED-TERMINAL VALUE 8.
    88 FDE-INVALID-FORM-IDENTIFIER VALUE 9.
    88 FDE-INVALID-USER-ENTRY VALUE 10.
    88 FDE-UNKNOWN-VARIABLE-NAME VALUE 11.

    88 FDE-TOO-MANY-INTEGERS VALUE 12.
    88 FDE-OBJECT-NAME-EXISTS VALUE 13.
    88 FDE-WORK-INVALID VALUE 14.
    88 FDE-INVALID-X-FORM-POSITION VALUE 15.
```

```
88 FDE-INVALID-Y-FORM-POSITION VALUE 16.
88 FDE-INVALID-WIDTH VALUE 17.
88 FDE-INVALID-HEIGHT VALUE 18.
88 FDE-INVALID-MESSAGE-FORM-NAME VALUE 19.

88 FDE-INVALID-OCCURRENCE VALUE 20.
88 FDE-INVALID-CHARACTER-POSITION VALUE 21.
88 FDE-INVALID-MODE VALUE 22.
88 FDE-INVALID-STATE VALUE 23.

88 FDE-INVALID-VARIABLE-VALUE VALUE 24.
88 FDE-INVALID-OBJECT-NAME VALUE 25.
88 FDE-INVALID-FORM-NAME VALUE 26.
88 FDE-FORM-CLOSED VALUE 27.

88 FDE-TOO-MANY-ATTRIBUTES VALUE 28.
88 FDE-INVALID-ATTRIBUTE-NAME VALUE 29.
88 FDE-TOO-MANY-SCREEN-OCCURRENCE VALUE 30.
88 FDE-NO-FORM-DEFINITION VALUE 31.

88 FDE-TOO-MANY-STORED-OCCURRENCE VALUE 32.
88 FDE-UNKNOWN-OBJECT-NAME VALUE 33.
88 FDE-NO-DEFINE-OBJECT-NAME VALUE 34.
88 FDE-INVALID-NAME VALUE 35.

88 FDE-SYSTEM-ERROR VALUE 36.
88 FDE-INVALID-TABLE-NAME VALUE 37.
88 FDE-INVALID-VARIABLE-NAME VALUE 38.
88 FDE-FORM-PUSHED VALUE 39.

88 FDE-UNKNOWN-TABLE-NAME VALUE 40.
88 FDE-NO-VARIABLE-DEFINED VALUE 41.
88 FDE-NO-FORMS-TO-POP VALUE 42.
88 FDE-ONLY-CHARACTER-DATA VALUE 43.

88 FDE-ONLY-NONCHARACTER-DATA VALUE 44.
88 FDE-FORM-DEFINITION-ERRORS VALUE 45.
88 FDE-NO-FORMS-TO-PUSH VALUE 46.
88 FDE-INVALID-PROGRAM-VALUES VALUE 47.
```

```
88 FDE-INPUT-HAS-UNKNOWN-VALUE VALUE 48.
88 FDE-INVALID-INPUT-VALUES VALUE 49.
88 FDE-NOT-AN-INPUT-VARIABLE VALUE 50.
88 FDE-CURSOR-NOT-IN-FORM VALUE 51.

88 FDE-FORM-HAS-NO-VARIABLES VALUE 52.
88 FDE-NO-FORMS-TO-SHOW VALUE 53.
88 FDE-FORM-NOT-SCHEDULED VALUE 54.
88 FDE-INVALID-EVENT-NAME VALUE 55.

88 FDE-INVALID-X-POSITION VALUE 56.
88 FDE-INVALID-Y-POSITION VALUE 57.
88 FDE-UNKNOWN-EVENT-NAME VALUE 58.
88 FDE-INVALID-DECK-NAME VALUE 59.

88 FDE-INVALID-RECORD-NAME VALUE 60.
88 FDE-OBJECT-EXISTS VALUE 61.
88 FDE-TABLE-NAME-EXISTS VALUE 62.
88 FDE-OBJECT-OVERLAYS VALUE 63.

88 FDE-TOO-MANY-REALS VALUE 64.
88 FDE-TOO-MANY-STRINGS VALUE 65.
88 FDE-NO-OBJECT-AT-POSITION VALUE 66.
88 FDE-ARRAY-TOO-SMALL VALUE 67.

88 FDE-STRING-TOO-SMALL VALUE 68.
88 FDE-VARIABLE-NAME-EXISTS VALUE 69.
88 FDE-FORM-ALREADY-ADDED VALUE 70.
88 FDE-INVALID-EVENT-ACTIVE VALUE 72.

88 FDE-CANNOT-UPDATE-OPENED-FORM VALUE 73.
88 FDE-HELP-FORM-EXISTS VALUE 74.
88 FDE-ERROR-FORM-EXISTS VALUE 75.
88 FDE-ERROR-MESSAGE-EXISTS VALUE 76.

88 FDE-HELP-MESSAGE-EXISTS VALUE 77.
88 FDE-INVALID-DISPLAY-NAME VALUE 78.
88 FDE-INVALID-REAL-RANGE VALUE 79.
88 FDE-INVALID-INTEGER-RANGE VALUE 80.
```

```
88 FDE-UNKNOWN-INTEGER-RANGE VALUE 81.
88 FDE-UNKNOWN-REAL-RANGE VALUE 82.
88 FDE-UNKNOWN-VALID-STRING VALUE 83.
88 FDE-DISPLAY-NAME-EXISTS VALUE 84.

88 FDE-EVENT-NAME-EXISTS VALUE 85.
88 FDE-UNKNOWN-DISPLAY-NAME VALUE 86.
88 FDE-TOO-MANY-FORM-NAMES VALUE 87.
88 FDE-TOO-MANY-FORM-OBJECTS VALUE 88.

88 FDE-NO-TEXT-AT-POSITION VALUE 89.
88 FDE-NO-TEXT-FOR-OBJECT VALUE 90.
88 FDE-UNKNOWN-OCCURRENCE VALUE 91.
88 FDE-NO-STRING VALUE 92.

88 FDE-RANGE-OVERLAP VALUE 93.
88 FDE-NO-COMMENTS-TO-DELETE VALUE 94.
88 FDE-OBJECT-OCCURRENCE-EXISTS VALUE 95.
88 FDE-NO-STRING-SPECIFIED VALUE 96.

88 FDE-VALID-STRING-EXISTS VALUE 97.
88 FDE-INVALID-OBJECT-CHANGE VALUE 98.
88 FDE-INVALID-ADDRESS VALUE 99.
88 FDE-TERMINAL-NOT-IDENTIFIED VALUE 100.

88 FDE-INVALID-FORM-LANGUAGE VALUE 101.
88 FDE-INVALID-FORM-AREA-KEY VALUE 102.
88 FDE-FORM-NAME-REQUIRED VALUE 103.
88 FDE-NO-FORMS-TO-READ VALUE 104.

88 FDE-INVALID-HELP-FORM-NAME VALUE 105.
88 FDE-INVALID-ERROR-FORM-NAME VALUE 106.
88 FDE-CREATE-MARK-INVALID VALUE 107.
88 FDE-DELETE-MARK-INVALID VALUE 108.

88 FDE-NO-MARK-DEFINED VALUE 109.
88 FDE-AREA-CUTS-OBJECT VALUE 110.
88 FDE-COPY-OUTSIDE-FORM VALUE 111.
88 FDE-MOVE-OUTSIDE-FORM VALUE 112.
```

```
88 FDE-INVALID-FORM-ATTRIBUTE VALUE 113.
88 FDE-INVALID-RECORD-ATTRIBUTE VALUE 114.
88 FDE-INVALID-OBJECT-KEY VALUE 115.
88 FDE-INVALID-OBJECT-ATTRIBUTE VALUE 116.

88 FDE-INVALID-TABLE-ATTRIBUTE VALUE 117.
88 FDE-PROGRAM-DATA-TYPE VALUE 118.
88 FDE-INVALID-OUTPUT-FORMAT-KEY VALUE 119.
88 FDE-INVALID-ERROR-KEY VALUE 120.

88 FDE-INVALID-VARIABLE-ATTRIBUTE VALUE 121.
88 FDE-INVALID-HELP-KEY VALUE 123.
88 FDE-FEATURE-NOT-IMPLEMENTED VALUE 124.
88 FDE-CANNOT-CHANGE-FORM VALUE 125.

88 FDE-INVALID-RECORD-TYPE VALUE 126.
88 FDE-OBJECT-NOT-IN-FORM VALUE 127.
88 FDE-INVALID-FORM-PROCESSOR VALUE 128.
88 FDE-INVALID-X-INCREMENT VALUE 129.

88 FDE-INVALID-Y-INCREMENT VALUE 130.
88 FDE-FORM-TOO-LARGE-FOR-SCREEN VALUE 131.
88 FDE-INVALID-TEXT-PROCESSING VALUE 132.
88 FDE-INVALID-DESIGN-FORM VALUE 133.

88 FDE-NO-OBJECT-VAR-DEFINED VALUE 134.
88 FDE-EVENT-NOT-ASSIGNED VALUE 135.
88 FDE-FORM-NOT-ENDED VALUE 136.
88 FDE-INVALID-EVENT-FORM-NAME VALUE 137.

88 FDE-INVALID-EVENT-FORM-KEY VALUE 138.
88 FDE-FORM-ALREADY-OPEN VALUE 139.
88 FDE-INVALID-EVENT-LABEL VALUE 140.
88 FDE-FORM-NEEDS-CONVERSION VALUE 141.

88 FDE-NO-EVENTS-ACTIVE VALUE 142.
88 FDE-DELETE-OUTSIDE-FORM VALUE 143.
88 FDE-MARK-OUTSIDE-FORM VALUE 144.
88 FDE-BAD-DATA-VALUE VALUE 145.
```

```
    88 FDE-RECORD-DEFN-NOT-WRITTEN VALUE 146.
    88 FDE-WRONG-VARIABLE-TYPE VALUE 147.
    88 FDE-INVALID-VARIABLE-LENGTH VALUE 148.
    88 FDE-EVENT-TRIGGER-EXISTS VALUE 149.

    88 FDE-FORM-ALREADY-COMBINED VALUE 150
    88 FDE-INVALID-TABLE-SIZE VALUE 151.
    88 FDE-FORM-NOT-ADDED-VALUE 152.
    88 FDE-INVALID-INPUT-FORMAT-KEY VALUE 153.
```

FDE$COBOL_VARIABLE_STATUS Deck

The contents of this deck follow.

```
01  FDE-COBOL-VARIABLE-STATUS USAGE COMP PIC S9(18) SYNC LEFT.
    88 FDE-NO-ERROR VALUE 0.
    88 FDE-INVALID-STRING VALUE 1.
    88 FDE-INVALID-REAL VALUE 2.

    88 FDE-INVALID-INTEGER VALUE 3.
    88 FDE-UNKNOWN-USER-VALUE VALUE 4.
    88 FDE-INVALID-BDP-DATA VALUE 5.
    88 FDE-NO-DIGITS VALUE 6.

    88 FDE-LOSS-OF-SIGNIFICANCE VALUE 7.
    88 FDE-VARIABLE-NO-FILLED VALUE 8.
    88 FDE-OVERFLOW VALUE 9.
    88 FDE-UNDERFLOW VALUE 10.

    88 FDE-INDEFINITE VALUE 11.
    88 FDE-INFINITE VALUE 12.
    88 FDE-VARIABLE-NOT-ENTERED VALUE 13.
    88 FDE-OUTPUT-FORMAT-BAD VALUE 14.
    88 FDE-VARIABLE-TRUNCATED VALUE 15.
```

Pascal Status Constants E

Pascal Status Constants

This section lists the values for the Pascal procedure STATUS parameter. You can copy the values into your program by using an SCU *COPY directive. The deck FDE$PASCAL_PROCEDURE_STATUS contains the values.

```
CONST
    fde$call_successful = 0;
    fde$terminal_disconnected = 1;
    fde$no_input_request = 2;
    fde$cursor_not_in_variable = 3;
    fde$more_errors_exist = 4;

    fde$unknown_form_name = 5;
    fde$form_compilation_errors = 6;
    fde$no_space_available = 7;
    fde$unsupported_terminal = 8;
    fde$invalid_form_identifier = 9;

    fde$invalid_user_entry = 10;
    fde$unknown_variable_name = 11;
    fde$too_many_integers = 12;
    fde$object_name_exists = 13;
    fde$work_area_invalid = 14;

    fde$invalid_x_form_position = 15;
    fde$invalid_y_form_position = 16;
    fde$invalid_width = 17;
    fde$invalid_height = 18;
    fde$invalid_message_form_name = 19;

    fde$invalid_occurrence = 20;
    fde$invalid_character_position = 21;
    fde$invalid_mode = 22;
    fde$invalid_state = 23;
    fde$invalid_variable_value = 24;

    fde$invalid_object_name = 25;
    fde$invalid_form_name = 26;
    fde$form_closed = 27;
    fde$too_many_attributes = 28;
    fde$invalid_attribute_name = 29;
```

```
fde$too_many_screen_occurrence = 30;
fde$no_form_definition = 31;
fde$too_many_stored_occurrence = 32;
fde$unknown_object_name = 33;
fde$no_define_object_name = 34;

fde$invalid_name = 35;
fde$system_error = 36;
fde$invalid_table_name = 37;
fde$invalid_variable_name = 38;
fde$form_pushed = 39;

fde$unknown_table_name = 40;
fde$no_table_variable_defined = 41;
fde$no_forms_to_pop = 42;
fde$only_character_data = 43;
fde$only_noncharacter_data = 44;

fde$form_definition_errors = 45;
fde$no_forms_to_push = 46;
fde$invalid_program_values = 47;
fde$input_has_unknown_value = 48;
fde$invalid_input_values = 49;

fde$not_an_input_variable = 50;
fde$cursor_not_in_form = 51;
fde$form_has_no_variables = 52;
fde$no_forms_to_show = 53;
fde$form_not_scheduled = 54;

fde$invalid_event_name = 55;
fde$invalid_x_position = 56;
fde$invalid_y_position = 57;
fde$unknown_event_name = 58;
fde$invalid_deck_name = 59;

fde$invalid_record_name = 60;
fde$object_exists = 61;
fde$table_name_exists = 62;
fde$object_overlays = 63;
fde$too_many_reals = 64;
```

```
fde$too_many_strings = 65;
fde$no_object_at_position = 66;
fde$array_too_small = 67;
fde$string_too_small = 68;
fde$variable_name_exists = 69;

fde$form_already_added = 70;
fde$invalid_event_active = 72;
fde$cannot_update_opened_form = 73;
fde$help_form_exists = 74;
fde$error_form_exists = 75;

fde$error_message_exists = 76;
fde$help_message_exists = 77;
fde$invalid_display_name = 78;
fde$invalid_real_range = 79;
fde$invalid_integer_range = 80;

fde$unknown_integer_range = 81;
fde$unknown_real_range = 82;
fde$unknown_valid_string = 83;
fde$display_name_exists = 84;
fde$event_name_exists = 85;

fde$unknown_display_name = 86;
fde$too_many_form_names = 87;
fde$too_many_form_objects = 88;
fde$no_text_at_position = 89;
fde$no_text_for_object = 90;

fde$unknown_occurrence = 91;
fde$no_string = 92;
fde$range_overlap = 93;
fde$no_comments_to_delete = 94;
fde$object_occurrence_exists = 95;

fde$no_string_specified = 96;
fde$valid_string_exists = 97;
fde$invalid_object_change = 98;
fde$invalid_address = 99;
fde$terminal_not_identified = 100;
```

```
fde$invalid_form_language = 101;
fde$invalid_form_area_key = 102;
fde$form_name_required = 103;
fde$no_forms_to_read = 104;
fde$invalid_help_form_name = 105;

fde$invalid_error_form_name = 106;
fde$create_mark_invalid = 107;
fde$delete_mark_invalid = 108;
fde$no_mark_defined = 109;
fde$area_cuts_object = 110;

fde$copy_outside_form = 111;
fde$move_outside_form = 112;
fde$invalid_form_attribute = 113;
fde$invalid_record_attribute = 114;
fde$invalid_object_key = 115;

fde$invalid_object_attribute = 116;
fde$invalid_table_attribute = 117;
fde$program_data_type = 118;
fde$invalid_output_format_key = 119;
fde$invalid_error_key = 120;

fde$invalid_variable_attribute = 121;
fde$invalid_help_key = 123;
fde$feature_not_implemented = 124;
fde$cannot_change_form = 125;
fde$invalid_record_type = 126;

fde$object_not_in_form  = 127;
fde$invalid_form_processor = 128;
fde$invalid_x_increment = 129;
fde$invalid_y_increment = 130;
fde$form_too_large_for_screen = 131;

fde$invalid_text_processing = 132;
fde$invalid_design_form = 133;
fde$no_object_variable_defined = 134;
fde$event_not_assigned = 135;
fde$form_not_ended = 136;
```

```
fde$invalid_event_form_name = 137;
fde$invalid_event_form_key = 138;
fde$form_already_open = 139;
fde$invalid_event_label = 140;
fde$form_requires_conversion = 141;

fde$no_events_active = 142;
fde$delete_outside_form = 143;
fde$mark_outside_form = 144;
fde$bad_data_value = 145;
fde$record_defn_not_written = 146;

fde$wrong_variable_type = 147;
fde$invalid_variable_length = 148;
fde$event_trigger_exists = 149;
fde$form_already_combined = 150;
fde$invalid_table_size = 151;

fde$form_not_added = 152;
fde$invalid_input_format_key = 153;
fde$system_error_message = 154;
fde$system_help_message = 155;
fde$system_bad_key_message = 156;
```

CYBIL Constants and Types F

Constants F-1

Types F-3

CYBIL Constants and Types F

This section lists the constants and types that are in each external reference routine. These constants and types are made available to your program through the inclusion of procedure definitions in the program. Procedure definitions are included using SCU *COPYC directives. For examples of how this is done, refer to the CYBIL chapters in this manual and the section on file interface procedures in the CYBIL for NOS/VE Usage manual.

Constants

```
fdc$max_character_position = fdc$maximum_record_length;
fdc$maximum_comment_length = fdc$maximum_text_length;
fdc$maximum_comments = 10000;
fdc$maximum_errors = 10000;

fdc$maximum_error_length = fdc$maximum_text_length;
fdc$maximum_events = 1000;
fdc$maximum_help_length = fdc$maximum_text_length;
fdc$maximum_form_identifier = 1000;

fdc$maximum_objects = 10000;
fdc$maximum_object_displays = 100;
fdc$maximum_occurrence = 1000;
fdc$maximum_record_length = osc$max_segment_length;

fdc$maximum_table_variables = 10000;
fdc$maximum_tables = 10000;
fdc$maximum_text_length = cyc$max_string_size;
fdc$maximum_valid_ranges = 10000;

fdc$maximum_valid_string = fdc$maximum_text_length;
fdc$maximum_valid_strings = 10000;
fdc$maximum_variable_length = fdc$maximum_record_length;
fdc$maximum_variables = fdc$maximum_objects;

fdc$maximum_x_position = 256;
fdc$maximum_y_position = 256;
fdc$message_form_name = 'FDM$MESSAGE_FORM                    ';
fdc$new_line_character = $char (31); {Unit separator}
```

```
fdc$system_coordinate_system = fdc$character_system;
fdc$system_currency_sign = '$';
fdc$system_decimal_point = '.';
fdc$system_design_table_name = 'DTBL';

fdc$system_design_variable_name = 'DVAR';
fdc$system_display_name = 'HIGHLIGHT';
fdc$system_error_message = 'Please correct.';
fdc$system_exponent_character = 'E';

fdc$system_form_processor = fdc$cybil_processor;
fdc$system_help_message = 'Please enter.';
fdc$system_input_format = fdc$character_input_format;
fdc$system_io_mode = fdc$terminal_input_output;

fdc$system_output_format = fdc$character_output_format;
fdc$system_occurrence = 1;
fdc$system_program_data_type = fdc$program_character_type;
fdc$system_record_type = fdc$program_data_type_record;

fdc$system_thousands_separator = ',';
fdc$system_unknown_entry = '?';
fdc$system_user_entry = fdc$must_enter;

mlc$min_exponent_style = 0,
mlc$max_exponent_style = 6;
```

Types

```
fdt$change_form_key = (fdc$add_display_definition,
   fdc$add_event, fdc$add_form_comment, fdc$delete_all_displays,
   fdc$delete_all_events, fdc$delete_display_defintion,
   fdc$delete_event, fdc$delete_form_comments,
   fdc$design_display_attribute, fdc$design_variable_name,
   fdc$event_form, fdc$form_area, fdc$form_display_attribute,
   fdc$form_help, fdc$form_language, fdc$form_name,
   fdc$form_processor, fdc$message_form, fdc$unused_form_entry,
   fdc$validate_variable_values);

fdt$change_object_key = (fdc$object_name, fdc$object_display,
   fdc$object_position, fdc$unused_object_entry,
   fdc$object_width, fdc$object_height, fdc$object_text,
   fdc$object_line_x_increment, fdc$object_line_y_increment,
   fdc$object_text_processing);

fdt$change_record_key = (fdc$record_deck_name, fdc$record_name,
   fdc$record_type, fdc$table_access, fdc$unused_record_entry);

fdt$change_table_key = (fdc$add_table_variable,
   fdc$delete_table_variable, fdc$new_table_name,
   fdc$stored_occurrence, fdc$unused_table_entry,
   fdc$visible_occurrence);

fdt$change_variable_key = (fdc$error_display,
   fdc$output_format, fdc$input_format, fdc$io_mode,
   fdc$terminal_user_entry, fdc$variable_length,
   fdc$add_valid_real_range, fdc$delete_valid_real_range,
   fdc$add_valid_integer_range, fdc$delete_valid_integer_range,
   fdc$add_valid_string, fdc$delete_valid_string,
   fdc$variable_help, fdc$variable_error, fdc$add_var_comment,
   fdc$delete_var_comments, fdc$unused_variable_entry,
   fdc$new_variable_name, fdc$process_as_event,
   fdc$unknown_entry_character, fdc$string_compare_rules,
   fdc$program_data_type);
```

```
fdt$character_position = 1 .. fdc$max_character_position;

fdt$comment = string ( * <= fdc$maximum_comment_length);

fdt$comment_length = 0 .. fdc$maximum_comment_length;

fdt$digits_in_exponent = mlt$exponent_style;

fdt$digits_right_decimal = 1 .. 19;

fdt$display_attribute = (fdc$inverse_video, fdc$low_intensity,
   fdc$high_intensity, fdc$blink, fdc$underline, fdc$protect,
   fdc$hidden, fdc$black_foreground, fdc$black_background,
   fdc$blue_foreground, fdc$blue_background,
   fdc$green_foreground, fdc$green_background,
   fdc$magenta_foreground, fdc$magenta_background,
   fdc$red_foreground, fdc$red_background, fdc$cyan_foreground,
   fdc$cyan_background, fdc$yellow_foreground,
   fdc$yellow_background, fdc$white_foreground,
   fdc$white_background, fdc$fine_line, fdc$medium_line,
   fdc$bold_line, fdc$fine_border, fdc$medium_border,
   fdc$bold_border, fdc$italic_display_attribute,
   fdc$title_display_attribute, fdc$input_display_attribute,
   fdc$error_display_attribute, fdc$message_display_attribute,
   fdc$display_left_to_right, fdc$display_right_to_left,
   fdc$push_input_character, fdc$user_attribute_1,
   fdc$user_attribute_2, fdc$user_attribute_3,
   fdc$user_attribute_4, fdc$user_attribute_5,
   fdc$user_attribute_6, fdc$user_attribute_7,
   fdc$user_attribute_8, fdc$user_attribute_9,
   fdc$user_attribute_10);

fdt$display_attribute_set = set of fdt$display_attribute;
```

```
fdt$error_definition = record
   case key: fdt$error_key of
   =  fdc$error_form =
      error_form: ost$name,
   =  fdc$error_message =
      p_error_message: ^fdt$error_message,
   =  fdc$no_error_response =
      ,
   =  fdc$system_default_error =
      ,
   casend
recend;

fdt$error_input_conversion = record
   occurence: fdt$occurrence,
   variable_name: ost$name,
recend;

fdt$error_invalid_value = record
   occurence: fdt$occurrence,
   variable_name: ost$name,
recend;

fdt$error_key = (fdc$error_form, fdc$error_message,
   fdc$no_error_response, fdc$system_default_error);

fdt$error_message = string ( * <= fdc$maximum_error_length);

fdt$error_message_length = 0 .. fdc$maximum_error_length;

fdt$error_no_table_object = record
   occurrence: fdt$occurrence,
   table_name: ost$name,
   variable_name: ost$name,
recend;

fdt$error_no_table_variable = record
   table_name: ost$name,
   variable_name: ost$name,
recend;
```

```
fdt$error_no_variable_object = record
   occurrence: fdt$occurrence,
   variable_name: ost$name,
recend;

fdt$error_output_conversion = record
   occurence: fdt$occurrence,
   variable_name: ost$name,
recend;

fdt$event_action = (fdc$return_program_normal,
   fdc$return_program_abnormal, fdc$page_table_forward,
   fdc$page_table_backward, fdc$scroll_table_forward,
   fdc$scroll_table_backward, fdc$display_help, fdc$erase_help,
   fdc$execute_command, fdc$ignore_event,
   fdc$tab_to_next_form_field, fdc$tab_to_previous_form_field,
   fdc$scroll_variable_forward, fdc$scroll_variable_backward),
   fdc$page_variable_forward,fdc$page_variable_backward,
   fdc$page_variable_first,fdc$page_variable_last,
   fdc$page_table_first,fdc$page_table_last);

fdt$event_command = string ( * );

fdt$event_form_definition = record
   case key: fdt$event_form_key of
   =  fdc$no_event_form =
      ,
   =  fdc$system_default_event_form =
      ,
   =  fdc$user_event_form =
      event_form_name: ost$name,
   casend
recend;
```

```
fdt$event_form_key = (fdc$no_event_form,
   fdc$system_default_event_form, fdc$user_event_form);

fdt$event_position = record
   form_identifier: fdt$form_identifier,
   form_x_position: fdt$x_position,
   form_y_position: fdt$y_position,
   screen_x_position: fdt$x_position,
   screen_y_position: fdt$y_position,
   case key: fdt$event_position_key of
   =  fdc$form_event =
      ,
   =  fdc$object_event =
      object_name: ost$name,
      object_occurrence: fdt$occurrence,
      object_x_position: fdt$x_position,
      object_y_position: fdt$y_position,
      case object_definition_key: fdt$object_definition_key of
      =  fdc$box,fdc$constant_text, fdc$constant_text_box,
         fdc$line,fdc$table =
         ,
      =  fdc$variable_text, fdc$variable_text_box =
        character_position: fdt$character_position,
      casend
   casend
recend;
```

```
fdt$event_position_key = (fdc$form_event, fdc$object_event,
   fdc$screen_event);

fdt$event_trigger = (fdc$next, fdc$help, fdc$stop, fdc$back,
   fdc$up, fdc$down, fdc$forward, fdc$backward, fdc$undo,
   fdc$redo, fdc$quit, fdc$exit, fdc$first, fdc$last, fdc$edit,
   fdc$data, fdc$function_1, fdc$function_2, fdc$function_3,
   fdc$function_4, fdc$function_5, fdc$function_6,
   fdc$function_7, fdc$function_8, fdc$function_9,
   fdc$function_10, fdc$function_11, fdc$function_12,
   fdc$function_13, fdc$function_14, fdc$function_15,
   fdc$function_16, fdc$shift_next, fdc$shift_help,
   fdc$shift_stop, fdc$shift_back, fdc$shift_up, fdc$shift_down,
   fdc$shift_forward, fdc$shift_backward, fdc$shift_edit,
   fdc$shift_data, fdc$shift_function_1, fdc$shift_function_2,
   fdc$shift_function_3, fdc$shift_function_4,
   fdc$shift_function_5, fdc$shift_function_6,
   fdc$shift_function_7, fdc$shift_function_8,
   fdc$shift_function_9, fdc$shift_function_10,
   fdc$shift_function_11, fdc$shift_function_12,
   fdc$shift_function_13, fdc$shift_function_14,
   fdc$shift_function_15, fdc$shift_function_16, fdc$pick,
   fdc$insert_line, fdc$delete_line, fdc$home_cursor,
   fdc$clear_screen, fdc$time_out, fdc$variable_trigger);

fdt$exponent_output_format = record
   field_width: fdt$real_field_width {w FORTRAN descriptor},
   digits_in_exponent: fdt$digits_in_exponent {e FORTRAN
      descriptor},
   digits_right_decimal: fdt$digits_right_decimal {d FORTRAN
      descriptor},
   sign_treatment: fdt$sign_treatment,
   suppress_zero: boolean {TRUE to display zero as blanks},
recend;

fdt$float_output_format = record
   digits_right_decimal: fdt$digits_right_decimal
      {d FORTRAN descriptor},
   field_width: fdt$real_field_width {w FORTRAN descriptor},
   sign_treatment: fdt$sign_treatment,
   suppress_zero: boolean {TRUE to display zero as blanks},
recend;
```

```
fdt$form_area = record
   case key: fdt$form_area_key of
   =  fdc$defined_area =
      x_position: fdt$x_position,
      y_position: fdt$y_position,
      width: fdt$width,
      height: fdt$height,
   =  fdc$screen_area =
      ,
   casend
recend;

fdt$form_area_key = (fdc$defined_area, fdc$screen_area);

fdt$form_attribute = record
   put_value_status: fdt$put_value_status {output},
   case key: fdt$change_form_key {input} of {input}
   =  fdc$add_event =
      event_name: ost$name,
      event_label: ost$name,
      event_trigger: fdt$event_trigger,
      case event_action: fdt$event_action of
      =  fdc$execute_command =
         p_event_command: ^fdt$event_command,
       casend,
   =  fdc$add_form_comment =
      p_form_comment: ^fdt$comment,
   =  fdc$add_display_definition =
      display_attribute: fdt$display_attribute_set,
      display_name: ost$name,
   =  fdc$delete_all_displays =
      ,
   =  fdc$delete_all_events =
      ,
   =  fdc$delete_event, fdc$delete_display_definition =
      name: ost$name,
   =  fdc$delete_form_comments =
      ,
   =  fdc$design_display_attribute =
      design_display_attribute: fdt$display_attribute_set,
   =  fdc$design_variable_name =
      design_variable_name: ost$name,
```

```
      =  fdc$event_form =
         event_form_definition: fdt$event_form_definition,
      =  fdc$form_area =
         form_area: fdt$form_area,
      =  fdc$form_display_attribute =
         form_display_attribute: fdt$display_attribute_set,
      =  fdc$form_help =
         form_help: fdt$help_definition,
      =  fdc$form_language =
         form_language: ost$natural_language,
      =  fdc$form_name =
         form_name: ost$name,
      =  fdc$form_processor =
         form_processor: fdt$form_processor,
      =  fdc$message_form =
         message_form: ost$name,
      =  fdc$unused_form_entry =
         ,
      =  fdc$validate_variable_values =
         validate_variable_values: boolean,
      casend
   recend;

   fdt$form_attributes = array [1 .. * ] of fdt$form_attribute;

   fdt$form_definition_error_key = (fdc$no_table_object,
      fdc$no_table_variable, fdc$no_variable_object,
      fdc$unequal_tbl_obj_width, fdc$no_variable_definition,
      fdc$error_input_conversion, fdc$error_output_conversion,
      fdc$error_invalid_value);

   fdt$form_identifier = 1 .. fdc$maximum_form_identifier;

   fdt$form_module = SEQ ( * );

   fdt$form_name = record
      name: ost$name,
      name_selection: fdt$name_selection,
   recend;

   fdt$form_names = array [1 .. * ] of fdt$form_name;
```

```
fdt$form_object = record
   name: ost$name,
   object: fdt$object_definition_key,
   occurrence: fdt$occurrence,
   x_position: fdt$x_position,
   y_position: fdt$y_position,
recend;

fdt$form_objects = array [1 .. * ] of fdt$form_object;

fdt$form_processor = (fdc$ansi_fortran_processor,
   fdc$cdc_fortran_processor, fdc$cobol_processor,
   fdc$cybil_processor, fdc$scl_processor);

fdt$get_error_definition = record
   case key: fdt$get_error_key of
   =  fdc$get_error_form =
      error_form: ost$name,
   =  fdc$get_error_message, fdc$get_system_default_error =
      error_message_length: fdt$error_message_length,
   =  fdc$get_no_error_response =
      ,
   casend
recend;

fdt$get_error_key = (fdc$get_error_form, fdc$get_error_message,
   fdc$get_no_error_response, fdc$get_system_default_error);

fdt$get_form_attribute = record
    get_value_status: fdt$get_value_status {output},
    case key: {input} fdt$get_form_key of
    =  fdc$get_event_command =
       event_command_name: {input} ost$name,
       p_event_command: {output} ^fdt$event_command,
    =  fdc$get_event_form =
       event_form_definition: {output} fdt$event_form_definition,
    =  fdc$get_event_form_identifier =
       event_form_identifier: {output} fdt$form_identifier,
    =  fdc$get_form_area =
       form_area: {output} fdt$form_area,
    =  fdc$get_form_comment_length =
       form_comment_length: {output} fdt$comment_length,
```

```
    = fdc$get_form_display_attribute =
      form_display_attribute: {output} fdt$display_attribute_set,
    = fdc$get_form_help =
      form_help: {output} fdt$get_help_definition,
    = fdc$get_form_help_message =
      p_form_help_message: {input} ^fdt$help_message,
    = fdc$get_form_language =
      form_language: {output} ost$natural_language,
    = fdc$get_form_name =
      form_name: {output} ost$name,
    = fdc$get_form_processor =
      form_processor: {output} fdt$form_processor,
    = fdc$get_message_form =
      message_form: {output} ost$name,
    = fdc$get_next_event =
      event_action: {output} fdt$event_action,
      event_label: {output} ost$name,
      event_name: {output} ost$name,
      event_command_length: {output} integer,
      event_trigger: {output} fdt$event_trigger,
    = fdc$get_next_form_comment =
      p_form_comment: {input} ^fdt$comment,
    = fdc$get_next_display =
      display_attribute: {output} fdt$display_attribute_set,
      display_name: {output} ost$name,
    = fdc$get_number_events =
      number_events: {output} fdt$number_events,
    = fdc$get_number_form_comments =
      number_form_comments: {output} fdt$number_comments,
    = fdc$get_number_displays =
      number_form_displays: {output} fdt$number_object_displays,
    = fdc$get_number_objects =
      number_objects: {output} fdt$number_objects,
    = fdc$get_number_tables =
      number_tables: {output} fdt$number_tables,
    = fdc$get_number_variables =
      number_variables: {output} fdt$number_variables,
    = fdc$get_unused_form_entry =
      ,
    casend
  recend;
```

```
fdt$get_form_key = (fdc$get_event_command, fdc$get_event_form,
   fdc$get_event_form_identifier, fdc$get_form_area,
   fdc$get_form_comment_length, fdc$get_form_display_attribute,
   fdc$get_form_help, fdc$get_form_help_message,
   fdc$get_form_language, fdc$get_form_name,
      fdc$get_form_processor,
   fdc$get_message_form, fdc$get_next_display,
      fdc$get_next_event,
   fdc$get_next_form_comment, fdc$get_number_displays,
   fdc$get_number_events, fdc$get_number_form_comments,
   fdc$get_number_objects, fdc$get_number_tables,
   fdc$get_number_variables, fdc$get_unused_form_entry);

fdt$get_form_attributes = array [1 .. * ] of
   fdt$get_form_attribute;

fdt$get_help_definition = record
   case key: fdt$get_help_key of
   =  fdc$get_help_form =
      help_form: ost$name,
   =  fdc$get_help_message, fdc$get_system_default_help =
      help_message_length: fdt$help_message_length,
   =  fdc$get_no_help_response =
      ,
   casend
recend;
```

```
fdt$get_object_attribute = record
   get_value_status: fdt$get_value_status {output},
   case key: {input} fdt$get_object_key of
   =  fdc$get_object_definition =
      get_object_definition: {output} fdt$get_object_definition,
   =  fdc$get_object_display =
      display_attribute: {output} fdt$display_attribute_set,
   =  fdc$get_object_name =
      object_name: {output} ost$name,
      occurrence: {output} fdt$occurrence,
   =  fdc$get_object_text =
      p_text: {input} ^fdt$text,
   =  fdc$get_object_text_length =
      text_length: {output} fdt$text_length,
   =  fdc$get_unused_object_entry =
      ,
   casend
recend;

fdt$get_object_attributes = array [1 .. * ] of
   fdt$get_object_attribute;

fdt$get_object_definition = record
   case key: {input} fdt$object_definition_key of
   =  fdc$box =
      box_width: {output} fdt$width,
      box_height: {output} fdt$height,
   =  fdc$line =
      x_increment: {output} fdt$x_increment,
      y_increment: {output} fdt$y_increment,
   =  fdc$constant_text =
      constant_text_width: {output} fdt$width,
      constant_text_length: {output} fdt$text_length,
   =  fdc$constant_text_box =
      constant_box_height: {output} fdt$height,
      constant_box_processing: {output} fdt$text_box_processing,
      constant_box_width: {output} fdt$width,
      constant_box_text_length: {output} fdt$text_length,
   =  fdc$table =
      table_height: {output} fdt$height,
      table_width: {output} fdt$width,
```

```
   =  fdc$variable_text_box =
      variable_box_height: {output} fdt$height,
      variable_box_processing: {output} fdt$text_box_processing,
      variable_box_text_length: {output} fdt$text_length,
      variable_box_width: {output} fdt$width,
   =  fdc$variable_text =
      variable_text_length: {output} fdt$text_length,
      variable_text_width: {output} fdt$width,
   casend
recend;

fdt$get_object_key = (fdc$get_object_definition,
   fdc$get_object_display, fdc$get_object_name,
   fdc$get_object_text, fdc$get_object_text_length,
   fdc$get_unused_object_entry);

fdt$get_record_attribute = record
   get_value_status {output} : fdt$get_value_status,
   case key {input} fdt$get_record_key of
   =  fdc$get_record_deck_name =
      record_deck_name: {output} ost$name,
   =  fdc$get_record_length =
      record_length {output} : fdt$record_length,
   =  fdc$get_record_name =
      record_name {output} : ost$name,
   =  fdc$get_record_type =
      record_type {output} : fdt$record_type,
   =  fdc$get_table_access =
      table_name {input} : ost$name,
      access_all_occurrences {output} : boolean,
   =  fdc$get_unused_record_entry =
      ,
    casend
recend;

fdt$get_record_attributes = array [1 .. * ] of
   fdt$get_record_attribute;
```

```
fdt$get_record_key = (fdc$get_number_record_variable,
   fdc$get_record_deck_name, fdc$get_record_definition,
   fdc$get_record_length, fdc$get_record_name,
   fdc$get_record_type, fdc$get_record_variable_names,
   fdc$get_table_access, fdc$get_unused_record_entry);

fdt$get_table_attribute = record
   get_value_status: {output} fdt$get_value_status,
   case key: {input} fdt$get_table_key of
   =  fdc$get_next_table_variable =
      variable_name: {output} ost$name,
   =  fdc$get_number_table_variables =
      number_table_variables: {output}
      fdt$number_table_variables,
   =  fdc$get_stored_occurrence =
      stored_occurrence: {output} fdt$occurrence,
   =  fdc$get_unused_table_entry =
      ,
   =  fdc$get_visible_occurrence =
      visible_occurrence: {output} fdt$occurrence,
   casend
recend;

fdt$get_table_attributes = array [1 .. * ] of
   fdt$get_table_attribute;

fdt$get_table_key = (fdc$get_next_table_variable,
   fdc$get_number_table_variables, fdc$get_stored_occurrence,
   fdc$get_unused_table_entry, fdc$get_visible_occurrence);

fdt$get_value_status = (fdc$system_computed_value,
   fdc$system_default_value, fdc$undefined_value,
   fdc$unprocessed_get_value, fdc$user_defined_value);

fdt$get_variable_attribute = record
   get_value_status: {output} fdt$get_value_status,
   case key: {input} fdt$get_variable_key of
   =  fdc$get_error_display =
      display_attribute: {output} fdt$display_attribute_set,
   =  fdc$get_input_format =
      input_format: {output} fdt$input_format,
   =  fdc$get_io_mode =
      io_mode: {output} fdt$io_mode,
```

```
=  fdc$get_next_valid_real_range =
   minimum_real: {output} real,
   maximum_real: {output} real,
=  fdc$get_next_valid_string =
   p_valid_string: {input} ^fdt$valid_string,
=  fdc$get_next_var_comment =
   p_var_comment: {input} ^fdt$comment,
=  fdc$get_number_valid_integers =
   number_valid_integers: {output} fdt$number_valid_integers,
=  fdc$get_number_valid_reals =
   number_valid_reals: {output} fdt$number_valid_reals,
=  fdc$get_number_valid_strings =
   number_valid_strings: {output} fdt$number_valid_strings,
=  fdc$get_number_var_comments =
   number_var_comments: {output} fdt$number_comments,
=  fdc$get_output_format =
   output_format: {output} fdt$output_format,
=  fdc$get_process_as_event =
   process_as_event: {output} boolean,
=  fdc$get_program_data_type =
   program_data_type: {output} fdt$program_data_type,
=  fdc$get_string_compare_rules =
   compare_in_upper_case: {output} boolean,
   compare_to_unique_substring: {output} boolean,
=  fdc$get_terminal_user_entry =
   terminal_user_entry: {output} fdt$terminal_user_entry,
=  fdc$get_unknown_entry_character =
   unknown_entry_character: {output} string (1),
=  fdc$get_unused_variable_entry =
   ,
=  fdc$get_valid_integer_range =
   minimum_integer: {output} integer,
   maximum_integer: {output} integer,
=  fdc$get_valid_string_length =
   valid_string_length: {output} fdt$valid_string_length,
=  fdc$get_var_comment_length =
   var_comment_length: {output} fdt$comment_length,
=  fdc$get_var_error_message =
   p_error_message: {input} ^fdt$error_message,
=  fdc$get_var_help_message =
   p_help_message: {input} ^fdt$help_message,
```

```
      =  fdc$get_variable_error =
         variable_error: {output} fdt$get_error_definition,
      =  fdc$get_variable_help =
         variable_help: {output} fdt$get_help_definition,
      =  fdc$get_variable_length =
         variable_length: {output} fdt$variable_length,
      casend
   recend;

   fdt$get_variable_attributes = array [1 .. * ] of
      fdt$get_variable_attribute;

   fdt$get_variable_key = (fdc$get_error_display,
      fdc$get_input_format, fdc$get_io_mode,
      fdc$get_next_valid_real_range, fdc$get_next_valid_string,
      fdc$get_next_var_comment, fdc$get_number_valid_integers,
      fdc$get_number_valid_reals, fdc$get_number_valid_strings,
      fdc$get_number_var_comments, fdc$get_output_format,
      fdc$get_process_as_event, fdc$get_program_data_type,
      fdc$get_string_compare_rules, fdc$get_terminal_user_entry,
      fdc$get_unknown_entry_character,
      fdc$get_unused_variable_entry, fdc$get_valid_integer_range,
      fdc$get_valid_string_length, fdc$get_var_comment_length,
      fdc$get_var_error_message, fdc$get_var_help_message,
      fdc$get_variable_help, fdc$get_variable_error,
      fdc$get_variable_length);

   fdt$height = 1 .. fdc$maximum_y_position;

   fdt$help_definition = record
      case key: fdt$help_key of
      =  fdc$help_form =
         help_form: ost$name,
      =  fdc$help_message =
         p_help_message: ^fdt$help_message,
      =  fdc$no_help_response, fdc$system_default_help =
         ,
      casend
   recend;
```

```
fdt$help_key = (fdc$help_form, fdc$help_message,
   fdc$no_help_response, fdc$system_default_help);

fdt$help_message = string ( * <= fdc$maximum_help_length);

fdt$help_message_length = 0 .. fdc$maximum_help_length;

fdt$input_currency_format = record
   currency_sybmol: string (1),
   thousands_separator: string (1),
   decimal_point: string (1),
recend;

fdt$input_format = record
   case key: fdt$input_format_key of
   =   fdc$character_input_format, fdc$alphabetic_input_format,
       fdc$digits_input_format, fdc$real_input_format,
       fdc$signed_input_format, fdc$ydm_format, fdc$mdy_format,
       fdc$dmy_format, fdc$iso_date_format,
       fdc$month_dd_yyyy_format =
       ,
    =  fdc$currency_input_format =
       input_currency_format: fdt$input_currency_format,
    casend
recend;

fdt$input_format_key = (fdc$alphabetic_input_format,
   fdc$character_input_format, fdc$currency_input_format,
   fdc$digits_input_format, fdc$dmy_format, fdc$mdy_format,
   fdc$month_dd_yyyy_format,fdc$iso_date_format,
   fdc$real_input_format, fdc$signed_input_format,
   fdc$ydm_format);

fdt$integer_field_width = 1 .. 19;

fdt$integer_output_format = record
   field_width: fdt$integer_field_width {w FORTRAN descriptor},
   minimum_output_digits: fdt$minimum_output_digits {m FORTRAN
      descriptor},
   sign_treatment: fdt$sign_treatment,
recend;
```

```
fdt$io_mode = (fdc$program_input_output {no io to terminal},
   fdc$terminal_input, fdc$terminal_input_output,
   fdc$terminal_output);

fdt$minimum_output_digits = 0 .. 19;

fdt$name_selection = (fdc$select_object, fdc$select_table,
   fdc$select_variable);

fdt$number_comments = 0 .. fdc$maximum_comments;

fdt$number_errors = integer;

fdt$number_events = 0 .. fdc$maximum_events;

fdt$number_names = integer;

fdt$number_object_displays = 0 .. fdc$maximum_object_displays;

fdt$number_objects = 0 .. fdc$maximum_objects;

fdt$number_table_variables = 0 .. fdc$maximum_table_variables;

fdt$number_tables = 0 .. fdc$maximum_tables;

fdt$number_valid_integers = 0 .. fdc$maximum_valid_ranges;

fdt$number_valid_reals = 0 .. fdc$maximum_valid_ranges;

fdt$number_valid_strings = 0 .. fdc$maximum_valid_strings;

fdt$number_variables = 0 .. fdc$maximum_variables;
```

```
fdt$object_attribute = record
   put_value_status: {output} fdt$put_value_status,
   case key: {input} fdt$change_object_key of
   =  fdc$object_display =
      display_attribute: {input} fdt$display_attribute_set,
   =  fdc$object_height =
      height: {input} fdt$height,
   =  fdc$object_line_x_increment =
      x_increment: {input} fdt$x_increment,
   =  fdc$object_line_y_increment =
      y_increment: {input} fdt$y_increment,
   =  fdc$object_name =
      object_name: {input} ost$name,
      occurrence: {input} fdt$occurrence,
   =  fdc$object_position =
      x_position: {input} fdt$x_position,
      y_position: {input} fdt$y_position,
   =  fdc$object_text =
      p_text: {input} ^fdt$text,
   =  fdc$object_text_processing =
      text_box_processing: {input} fdt$text_box_processing,
   =  fdc$object_width =
      width: {input} fdt$width,
   =  fdc$unused_object_entry =
      ,
   casend
recend;

fdt$object_attributes = array [1 .. * ] of fdt$object_attribute;

fdt$object_definition = record
   case key: {input} fdt$object_definition_key of {input}
   =  fdc$box =
      box_width: fdt$width,
      box_height: fdt$height,
   =  fdc$constant_text =
      constant_text_width: fdt$width,
      p_constant_text: ^fdt$text,
```

```
      =  fdc$constant_text_box =
         constant_box_height: fdt$height,
         constant_box_processing: fdt$text_box_processing,
         constant_box_width: fdt$width,
         p_constant_box_text: ^fdt$text,
      =  fdc$line =
         x_increment: fdt$x_increment,
         y_increment: fdt$y_increment,
      =  fdc$table =
         table_width: fdt$width,
         table_height: fdt$height,
      =  fdc$variable_text =
         p_variable_text: ^fdt$text,
         variable_text_width: fdt$width,
      =  fdc$variable_text_box =
         p_variable_box_text: ^fdt$text,
         variable_box_height: fdt$height,
         variable_box_processing: fdt$text_box_processing,
         variable_box_width: fdt$width,
      casend
   recend;

   fdt$object_definition_key = (fdc$box, fdc$constant_text,
      fdc$constant_text_box, fdc$line, fdc$table,
      fdc$variable_text, fdc$variable_text_box);

   fdt$object_event_position = record
      form_identifier: fdt$form_identifier,
      object_name: ost$name,
      occurrence: fdt$occurrence,
      case key: fdt$object_definition_key of
      =  fdc$box, fdc$line, fdc$constant_text,
         fdc$constant_text_box =
         {The x, y positions are relative to the form}
         form_x_position: fdt$x_position,
         form_y_position: fdt$y_position,
      =  fdc$variable_text, fdc$variable_text_box =
         character_position: fdt$character_position,
      casend
   recend;
```

```
fdt$occurrence = 1 .. fdc$maximum_occurrence;

fdt$output_currency_format = record
   currency_sybmol: string (1),
   thousands_separator: string (1),
   decimal_point: string (1),
   field_width: fdt$text_length,
   sign_treatment: fdt$sign_treatment,
   suppress_leading_zeros: boolean {TRUE to suppress},
recend;

fdt$output_format = record
   case key: fdt$output_format_key of
   =  fdc$character_output_format =
      ,
   =  fdc$currency_output_format =
      output_currency_format: fdt$output_currency_format,
   = fdc$dmy_output_format =
      {Uses an 8 character field, dd/mm/yy}
      ,
   =  fdc$e_e_output_format, fdc$g_e_output_format =
      exponent_output_format: fdt$exponent_output_format,
   =  fdc$f_output_format, fdc$e_output_format,
      fdc$g_output_format =
      float_output_format: fdt$float_output_format,
   =  fdc$integer_output_format =
      integer_output_format: fdt$integer_output_format,
   =  fdc$iso_output_format =
      {Uses a 10 character field, yyyy-mm-dd}
      ,
   =  fdc$mdy_output_format =
      {Uses an 8 character field, mm/dd/yy}
      ,
   =  fdc$month_dd_yyyy_out_format =
      {Uses a 18 character field, monthxxxx dd, yyyy}
      ,
   =  fdc$undefined_output_format =
      ,
   =  fdc$ydm_output_format =
      {Uses an 8 character field, yy/dd/mm}
      ,
   casend
recend;
```

```
fdt$output_format_key = (fdc$character_output_format,
   fdc$currency_output_format, fdc$dmy_output_format,
   fdc$e_e_output_format, fdc$e_output_format,
   fdc$f_output_format, fdc$g_e_output_format,
   fdc$g_output_format, fdc$iso_output_format,
   fdc$mdy_output_format, fdc$month_dd_yyyy_out_format,
   fdc$integer_output_format, fdc$undefined_output_format,
   fdc$ydm_output_format);

fdt$put_value_status = (fdc$put_value_accepted,
   fdc$unprocessed_put_value);

fdt$program_data_type = (fdc$program_character_type,
   fdc$program_integer_type, fdc$program_real_type,
   fdc$program_upper_case_type);

fdt$real_field_width = 1 .. 19;

fdt$record_attribute = record
   put_value_status: {output} fdt$put_value_status,
   case key: {input} fdt$change_record_key of
   =  fdc$record_deck_name =
      record_deck_name: {input} ost$name,
   =  fdc$record_name =
      record_name: {input} ost$name,
   =  fdc$record_type =
      record_type: {input} fdt$record_type,
   =  fdc$table_access =
      table_name: {input} ost$name,
      access_all_occurrences: {input} boolean,
   =  fdc$unused_record_entry =
      ,
   casend
recend;

fdt$record_attributes = array [1 .. * ] of fdt$record_attribute;

fdt$record_length = 0 .. fdc$maximum_record_length;

fdt$record_position = 1 .. fdc$maximum_record_length;
```

```
fdt$record_type = (fdc$character_record,
   fdc$program_data_type_record);

fdt$sign_treatment = mlt$sign_treatment;

fdt$table_attribute = record
   put_value_status: {output} fdt$put_value_status,
   case key: {input} fdt$change_table_key of
   =  fdc$add_table_variable, fdc$delete_table_variable =
      variable_name: {input} ost$name,
   =  fdc$new_table_name =
      new_table_name: {input} ost$name,
   =  fdc$stored_occurrence =
      stored_occurrence: {input} fdt$occurrence,
   =  fdc$unused_table_entry =
      ,
   =  fdc$visible_occurrence =
      visible_occurrence: {input} fdt$occurrence,
   casend
recend;

fdt$table_size = 0 .. fdc$maximum_occurrence;

fdt$table_attributes = array [1 .. * ] of fdt$table_attribute;

fdt$terminal_user_entry = set of (fdc$entry_optional,
   fdc$must_enter,fdc$may_enter_unknown, fdc$must_fill);

fdt$text = string ( * <= fdc$maximum_text_length);

fdt$text_box_processing = (fdc$center_characters,
   fdc$wrap_characters, fdc$wrap_words);

fdt$text_length = 0 .. fdc$maximum_text_length;

fdt$valid_string = string ( * <= fdc$maximum_valid_string);

fdt$valid_string_length = 0 .. fdc$maximum_valid_string;
```

```
fdt$variable_attribute = record
   put_value_status: {output} fdt$put_value_status,
   case key: {input} fdt$change_variable_key {input} of
   =  fdc$add_valid_integer_range,
      fdc$delete_valid_integer_range =
      maximum_integer: integer,
      minimum_integer: integer,
   =  fdc$add_valid_real_range, fdc$delete_valid_real_range =
      maximum_real: real,
      minimum_real: real,
   =  fdc$add_valid_string, fdc$delete_valid_string =
      p_valid_string: ^fdt$valid_string,
   =  fdc$add_var_comment =
      p_var_comment: ^fdt$comment,
   =  fdc$delete_var_comments =
      ,
   =  fdc$input_format =
      input_format: fdt$input_format,
   =  fdc$io_mode =
      io_mode: fdt$io_mode,
   =  fdc$new_variable_name =
      new_variable_name: ost$name,
   =  fdc$error_display =
      display_attribute: fdt$display_attribute_set,
   =  fdc$output_format =
      output_format: fdt$output_format,
   =  fdc$program_data_type =
      program_data_type: fdt$program_data_type,
   =  fdc$process_as_event =
      process_as_event: boolean {If true, the value of the
      variable is treated as an event rather than a data item to
      be transferred to and from a program},
   =  fdc$string_compare_rules =
      compare_in_upper_case: boolean,
      compare_to_unique_substring: boolean,
   =  fdc$terminal_user_entry =
      terminal_user_entry: fdt$terminal_user_entry,
   =  fdc$unknown_entry_character =
      unknown_entry_character: string (1),
   =  fdc$unused_variable_entry =
      ,
   =  fdc$variable_error =
      variable_error: fdt$error_definition,
```

```
    =  fdc$variable_help =
       variable_help: fdt$help_definition,
    =  fdc$variable_length =
       variable_length: fdt$variable_length,
    casend
recend;

fdt$variable_attributes = array [1 .. * ] of
    fdt$variable_attribute;

fdt$variable_length = 1 .. fdc$maximum_variable_length;

fdt$variable_status = (fdc$no_error, fdc$invalid_string,
    fdc$invalid_real, fdc$invalid_integer,
    fdc$unknown_user_value, fdc$invalid_bdp_data, fdc$no_digits,
    fdc$loss_of_significance, fdc$variable_not_filled,
    fdc$overflow, fdc$underflow, fdc$indefinite, fdc$infinite,
    fdc$variable_not_entered, fdc$output_format_bad,
    fdc$variable_truncated);

fdt$width = 1 .. fdc$maximum_x_position;

fdt$work_area_length = 1 .. fdc$maximum_record_length;

fdt$x_increment = 0 .. fdc$maximum_x_position - 1;

fdt$x_position = 1 .. fdc$maximum_x_position;

fdt$y_increment = 0 .. fdc$maximum_y_position - 1;

fdt$y_position = 1 .. fdc$maximum_y_position;

ost$name = string (osc$max_name_size);

ost$status = record
  case normal: boolean of
  = FALSE =
    condition: ost$status_condition_code,
    text: ost$string
  = TRUE =
    ,
  casend
recend;
```

```
mlt$exponent_style = mlc$min_expenent_style ..
   mlc$max_exponent_style;

mlt$sign_treatment = (mlc$minus_if_negative,
   mlc$always_signed);
```

FORTRAN Call Definitions G

FORTRAN Call Definitions G

The following FORTRAN call definitions give the aliases for the Screen Formatting subroutines used in the FORTRAN calls. These definitions must be present whenever you call Screen Formatting. Include the following SCU directive in every program or subroutine that has Screen Formatting calls:

*COPY FDP$FORTRAN_ALIASES

The contents of FDF$FORTRAN_ALIASES follows.

```
C$   EXTERNAL (ALIAS='FDP$XADD_FORM',LANG=FTN), FDADD
C$   EXTERNAL (ALIAS='FDP$XCHANGE_TABLE_SIZE',LANG=FTN), FDCHAT
C$   EXTERNAL (ALIAS='FDP$XCOMBINE_FORM',LANG=FTN), FDCOM
C$   EXTERNAL (ALIAS='FDP$XCLOSE_FORM',LANG=FTN), FDCLOS
C$   EXTERNAL (ALIAS='FDP$XDELETE_FORM',LANG=FTN), FDDEL
C$   EXTERNAL (ALIAS='FDP$XGET_INTEGER VARIABLE,LANG=FTN), FDGETI
C$   EXTERNAL (ALIAS='FDP$XGET_NEXT_EVENT',LAMG=FTN), FSGETE
C$   EXTERNAL (ALIAS='FDP$XGET_REAL_VARIABLE',LANG=FTN), FDGETR
C$   EXTERNAL (ALIAS='FDP$XGET_RECORD',LANG=FTN), FDGET
C$   EXTERNAL (ALIAS='FDP$XGET_STRING_VARIABLE',LANG=FTN), FDGETS
C$   EXTERNAL (ALIAS='FDP$XOPEN_FORM',LANG=FTN), FDOPEN
C$   EXTERNAL (ALIAS='FDP$XPOP_FORMS',LANG=FTN), FDPOP
C$   EXTERNAL (ALIAS='FDP$XPOSITION_FORM',LANG=FTN), FDPOS
C$   EXTERNAL (ALIAS='FDP$XPUSH_FORMS',LANG=FTN), FDPUSH
C$   EXTERNAL (ALIAS='FDP$XREAD_FORMS',LANG=FTN), FDREAD
C$   EXTERNAL (ALIAS='FDP$XREPLACE_INTEGER_VARIABLE',LANG=FTN), FDREPI
C$   EXTERNAL (ALIAS='FDP$XREPLACE_REAL_VARIABLE',LANG=FTN), FDREPR
C$   EXTERNAL (ALIAS='FDP$XREPLACE_RECORD'LANG=FTN), FDREP
C$   EXTERNAL (ALIAS='FDP$XREPLACE_STRING_VARIABLE',LANG=FTN), FDREPS
C$   EXTERNAL (ALIAS='FDP$XRESET_FORM',LANG=FTN), FDRESF
C$   EXTERNAL (ALIAS='FDP$XRESET_OBJECT_ATTRIBUTE',LANG=FTN), FDRESO
C$   EXTERNAL (ALIAS='FDP$XSET_CURSOR_POSITION',LANG=FTN), FDSETC
C$   EXTERNAL (ALIAS='FDP$XSET_LINE_MODE',LANG=FTN), FDSETL
C$   EXTERNAL (ALIAS='FDP$XSET_OBJECT_ATTRIBUTE',LANG=FTN), FDSETO
C$   EXTERNAL (ALIAS='FDP$XSHOW_FORMS',LANG=FTN), FDSHOW
```

Accessing Online Examples H

Accessing Examples by Name or by Manual H-2

Searching for Examples by Command or Procedure Name H-3

Viewing, Copying, Printing, and Executing Examples H-4

Using Function Keys and Directives H-5

Accessing Online Examples H

An online manual named Examples contains examples which show you how to use various NOS/VE concepts, SCL commands, and CYBIL procedures. You can use the online Examples manual to perform the following operations.

- Access examples by name, manual, command name, or procedure name.
- View the example.
- Print the example.
- Copy the example into your $USER catalog for subsequent reference or execution.
- Execute the example.

To access the online manual, enter:

```
/help manual=examples
```

In response, the system displays a menu of the topics for which examples are provided. This menu includes topics from the following manuals:

Ada for NOS/VE
COBOL for NOS/VE
CYBIL File Management
CYBIL Keyed-File and Sort/Merge Interfaces
CYBIL Language Definition
CYBIL Sequential and Byte-Addressable Files
CYBIL System Interface
FORTRAN for NOS/VE
Introduction to NOS/VE
NOS/VE File Editor
NOS/VE Screen Formatting
NOS/VE System Usage
NOS/VE Object Code Management
NOS/VE Source Code Management
Pascal for NOS/VE

Accessing Examples by Name or by Manual

In each of the printed manuals containing examples, the example's name is supplied in the introduction to the example. Because the online Examples manual is indexed by example name, you can access the example directly by specifying its name.

For example, suppose you are reading the CREATE_PERMIT_PF_1 example in the CYBIL File Management manual and you want to have a copy of the example in one of your catalogs. You can quickly access the example by using either of the following methods.

- Specify the name of the example on the SUBJECT parameter of the HELP command when you access the manual. For example:

  ```
  /help subject=create_permit_pf_1 manual=examples
  ```

- If you have already accessed the Examples manual, enter the example's name followed by a question mark:

  ```
  create_permit_pf_1?
  ```

You are then positioned to the introductory screen of the CREATE_PERMIT_PF_1 example. This screen prompts you to view, copy, or print the example.

To access examples associated with a specific manual, select an option from the main menu. The system displays a list of example names associated with that manual. You can then choose a specific example from the list.

Searching for Examples by Command or Procedure Name

The online Examples manual also enables you to search for examples by SCL command or CYBIL procedure names. You can either view the list of index topics by pressing the key associated with the Index operation, or you can access a topic directly by entering the command or procedure name itself.

For example, if you want to look at one or more ways in which the CREATE_FILE command is used, enter the following request on the home line:

```
create_file?
```

If you want to see one or more ways that the FSP$OPEN_FILE procedure call is used in examples, enter:

```
fsp$open_file?
```

In response, the system displays an example that illustrates the use of the procedure or command you specified.

You can also specify the command or procedure name on the SUBJECT parameter of the HELP command when you access the manual. For example:

```
/help subject=fsp$open_file manual=examples
```

To view a further example that illustrates the use of the command or procedure you specified, enter another question mark (?). You can enter as many question marks as there are examples indexed for that command or procedure.

When the number of examples for that command or procedure is exhausted, an informative message is displayed.

Viewing, Copying, Printing, and Executing Examples

After you access a particular example, the following menu of options appears:

```
Enter your menu choice:
 a. view the example
 b. copy the example
 c. print the example
d. execute the example
```

Use the menu of options as follows:

- To view the example, choose menu selection A, followed by a return. The example is displayed at your terminal. Since the example appears in full-screen mode, you can easily move from screen to screen by following the function key prompts.
- To copy the example to a file, choose menu selection B, followed by a return. You are then prompted for the name of the file to which you want the example copied. Once you enter a file name, NOS/VE displays a message verifying the name of the file to which the example was copied.
- To print the example, choose menu selection C. A message soon appears which indicates that the file has been sent to the printer.
- To execute the example, choose item D. The example is copied to a temporary file and executed.

Using Function Keys and Prompts

Once you access the online Examples manual, you can read it by pressing function keys and entering text in response to function key prompts, or entering menu choices on the home line.

Function key prompts for using this manual are displayed at the bottom of your screen, provided you are in full-screen mode. These function keys vary according to the type of terminal you are using.

If you need assistance on what a particular function key does, press the help key for your terminal, and then press the function key in question. Pressing the help key again displays a menu of online help options (such as how to use the menus, or how to page forward and backward).

The following function key prompts help you search for examples:

Function Key Prompt	Description
Find	Enables you to locate screens where an example, command, or procedure you specify appears.
Index	Enables you to access the manual's index. After pressing the key associated with this operation, you can do one of the following: • Specify the topic where you want to begin reading the index. • Press RETURN to display the beginning of the index.

Many terminals have function keys or dedicated keys that return you to the main menu (the first screen in the manual). On a VT220 terminal, hold down the shift key and press the **F17** key. Alternatively, you can enter the FIRST or TOP directive on the home line of any terminal at which you can read online manuals.

The Quit function key prompt is associated with the key(s) you press to leave the Examples manual. On a VT220 terminal, press the **F11** key. Alternatively, you can enter the QUIT directive on the home line of any terminal at which you can read online manuals.

Index

Index

A

Abnormal task 7-12
ADD_FORM 2-37
ADDF 2-37
Adding a form
  COBOL 3-36
  CYBIL 6-31
  FORTRAN 4-30
  Pascal 5-33
  SCL 2-37
Aliases, FORTRAN G-1
Alphabetic character A-1
Application prototype
  COBOL 3-8
  CYBIL 6-7
  FORTRAN 4-7
  Pascal 5-9
  SCL 2-8
Attributes
  Data 7-4
  Data validation 7-6
  Form 7-2, 43
  Form definition record 7-76
  Glossary definition A-1
  Object 7-78
  Output formatting 7-5
  Resetting
    COBOL 3-80
    CYBIL 6-66
    FORTRAN 4-71
    Pascal 5-66
    SCL 2-60
  Setting
    COBOL 3-85
    CYBIL 6-70
    FORTRAN 4-77
    Pascal 5-71
    SCL 2-60
  Table 7-73
  Terminal definition C-1
  Variable 7-3, 60

B

Batch mode A-1
Box 7-5, 9

C

Calling Screen Formatting
  COBOL 3-5
  CYBIL 6-4
  FORTRAN 4-4
  Pascal 5-6
  SCL 2-3
Catalog A-1
Catalog name A-1
CHANGE_TABLE_SIZE 2-38
Changing
  Form 7-24, 31, 86, 124
  Form definition record 7-87
  Form definition record attributes 7-76
  General form attributes 7-44
  Object attributes 7-78, 88
  Stored object 7-89
  Table attributes 7-73, 90
  Table size
    COBOL 3-38
    CYBIL 6-32
    FORTRAN 4-32
    Pascal 5-34
    SCL 2-38
  Variable attributes 7-60, 91
Character
  Data type attribute 7-4
  Glossary definition A-1
  Validation 7-6
  Wrap 7-5
CHATS 2-38
Circle form example
  COBOL 3-12
  CYBIL 6-11
  FORTRAN 4-11
  Pascal 5-13
  SCL 2-16
CLOF 2-40
CLOSE_FORM 2-40

Closing a form
  COBOL 3-40
  CYBIL 6-34
  FORTRAN 4-34
  Pascal 5-36
  SCL 2-40
COBOL
  Displaying forms 3-1
  Parameter definitions D-1
  Program 3-16
  Status checking 3-35
  Subroutines 3-35
COMBINE_FORM 2-41
Combining forms
  COBOL 3-41
  CYBIL 6-35
  Events 7-10
  FORTRAN 4-35
  Pascal 5-37
  SCL 2-41
COMF 2-41
Compiling a program
  COBOL 3-30
  CYBIL 6-25
  FORTRAN 4-24
  Pascal 5-27
Constant text objects
  Creating 7-99
  Definition 7-3
Constants
  CYBIL F-1
  Pascal E-1
Content validation 7-6
Conventions 10
Converting
  Program data 7-94
  User data 7-92
Copying
  Data definitions
    COBOL 3-4
    CYBIL 6-3
    FORTRAN 4-3
    Pascal 5-4
  Form 7-98
  Form definition decks
    COBOL 3-15
    CYBIL 6-14
    FORTRAN 4-13
    Pascal 5-16
  Objects 7-96
  Parameter definitions
    COBOL 3-3
  Procedure definitions
    CYBIL 6-2
    Pascal 5-2
  Text 7-96
Creating
  Constant text objects 7-99
  Design
    Form 7-100
    Text 7-102
  Error forms 7-16
  Event form 7-104
  Form definition record 7-42
  Forms
    Discussion 7-19
    Example 1-2, 6
    Procedure 7-103
    Using CYBIL 7-19
    Using SDF - See SDF manual
  General form attributes 7-44
  Help forms 7-16
  Mark display attribute 7-106
  Object 7-108
  Object attributes 7-78
  Program example 7-34
  Stored object 7-113
  Table 7-115
  Table attributes 7-73
  Variable attributes 7-60
  Variables
    CYBIL 7-116
    SCL 2-31
Currency format 7-62, 63
Cursor position
  COBOL 3-82
  CYBIL 6-67
  FORTRAN 4-73
  Moving 7-14
  Pascal 5-68
  SCL 2-59
CYBIL
  Constants and types F-1
  Creating forms 7-1, 85
  Displaying forms 6-1, 30
  Program for creating forms 7-31
  Program for displaying forms 6-15

Usage 1-3

D

Data
  Converting 7-92, 94
  Flow 7-3
  Type 7-4
  Validation attributes 7-6
Data definitions
  Copying
    COBOL 3-4
    CYBIL 6-3
    FORTRAN 4-3
    Pascal 5-4
Deactivate events
  COBOL 3-66
  CYBIL 6-54
  FORTRAN 4-59
  Pascal 5-56
  SCL 2-52
Defining
  Constant text objects 7-3
  Display attributes 7-15
  Events 7-9
  Form 7-2
  General form attributes 7-44
  Object text 7-3
  Tasks for events 7-10
  Variable attributes 7-3
  Variable text objects 7-3
DELETE_FORM 2-43
Deleting
  Form
    COBOL 3-43
    CYBIL 6-37
    FORTRAN 4-37
    Pascal 5-39
    SCL 2-43
  Mark display attribute 7-119
  Objects 7-118, 120
  Scheduled forms
    COBOL 3-63
    CYBIL 6-51
    FORTRAN 4-56
    Pascal 5-53
    SCL 2-49
  Stored Object 7-121
  Table 7-122
  Text 7-118
  Variable 7-123
DELF 2-43
Design form A-2
Design specification
  Creating 7-20
  Definition 1-11
  Usage 1-11
  Using
    COBOL 3-13
    CYBIL 6-12
    FORTRAN 4-12
    Pascal 5-14
    SCL 2-9, 21
Designing forms
  Dynamically 7-22
  Interactively 7-25
  Introduction 1-2
Digit A-2
Display attributes
  Changing 7-51
  Defining 7-15
  Definition 7-2
  Specifying 7-54
Displaying forms
  COBOL 3-5, 67
  CYBIL 6-4, 55
  Description 7-33
  FORTRAN 4-4, 60
  Pascal 5-6, 57
  SCL 2-4, 54

E

Editing a form 7-124
Ending a form definition 7-125
Error messages
  Default form 7-17
  Erasing 7-13
  Form attribute 7-55
  Information 7-16
  Validating data 7-6
Event form
  Creating 7-104
  Definitions 7-52
  Information 7-9
Event_label 7-44
$EVENT_NAME 2-64
Event_name 7-44

$EVENT_NORMAL 2-65
$EVENT_POSITION 2-66
Event_trigger 7-45
Events
  Abnormal
    COBOL 3-8
    CYBIL 6-7
    FORTRAN 4-7
    Pascal 5-9
    SCL 2-7
  Deactivate
    COBOL 3-66
    CYBIL 6-54
    FORTRAN 4-59
    Pascal 5-56
    SCL 2-52
  Defining 7-9
  Definition 1-10; A-2
  Form 7-9
  Getting the next
    COBOL 3-48
    CYBIL 6-40
    FORTRAN 4-42
    Pascal 5-42
    SCL 2-64
  Label 7-44
  Name 1-10; 7-44
  Normal
    COBOL 3-7
    CYBIL 6-6
    FORTRAN 4-6
    Pascal 5-8
    SCL 2-6
  Processing 1-10
  Requirements 1-10
  Specifying form
   definitions 7-52
  Standard 7-15, 45
  Trigger 7-45; C-2
Example
  Circle form
    COBOL 3-12
    CYBIL 6-11
    FORTRAN 4-11
    Pascal 5-13
    SCL 2-16
  Program
    COBOL 3-16
    CYBIL 6-15
    FORTRAN 4-15
    Pascal 5-17
    SCL 2-22
  Rectangle form
    COBOL 3-11
    CYBIL 6-10
    FORTRAN 4-10
    Pascal 5-12
    SCL 2-13
  Select form
    COBOL 3-10
    CYBIL 6-9
    FORTRAN 4-9
    Pascal 5-11
    SCL 2-12
Expanding a program
  COBOL 3-30
  CYBIL 6-25
  FORTRAN 4-24
  Pascal 5-27
  SCL 2-28

F

Family A-2
Family name A-2
File A-2
File name A-2
Form
  Adding
    COBOL 3-36
    CYBIL 6-31
    FORTRAN 4-30
    Pascal 5-33
    SCL 2-37
  Attributes 7-43, 126
  Changing 7-24, 86, 124
  Closing
    COBOL 3-40
    CYBIL 6-34
    FORTRAN 4-34
    Pascal 5-36
    SCL 2-40
  Combining
    COBOL 3-41
    CYBIL 6-35
    FORTRAN 4-35
    Pascal 5-37
    SCL 2-41
  Contents 7-2

Copying 7-98
Creating 1-6; 7-1, 19, 22, 103
Creating design 7-100
Deactivate events
COBOL 3-66
CYBIL 6-54
FORTRAN 4-59
Pascal 5-56
SCL 2-52
Definition of 7-2
Definition record 7-87
Deleting
COBOL 3-43, 63
CYBIL 6-37, 51
FORTRAN 4-37, 56
Pascal 5-39, 53
SCL 2-43, 49
Design
Dynamically 7-22
Interactively 7-25
Introduction 1-2
Display attributes 7-54, 57
Displaying
COBOL 3-67
CYBIL 6-55
Description 7-33
FORTRAN 4-60
Pascal 5-57
SCL 2-54
Ending a definition 7-125
Event 7-104
Example of creating 1-6; 7-34
Graphic object 1-6
Help 7-13
Interaction with a program 7-9
Managing
Example 1-6
Introduction 1-2
Multiple 7-9
Names 7-127
Objects 7-3, 128
Opening
COBOL 3-61
CYBIL 6-49
FORTRAN 4-54
Pascal 5-51
SCL 2-48
Popping
COBOL 3-63
CYBIL 6-51
FORTRAN 4-56
Pascal 5-53
SCL 2-49
Positioning
COBOL 3-64
CYBIL 6-52
FORTRAN 4-57
Pascal 5-54
SCL 2-50
Processor 7-55
Reading
COBOL 3-67
CYBIL 6-55
FORTRAN 4-60
Pascal 5-57
SCL 2-54
Record
COBOL 3-55, 75
CYBIL 6-45, 61
FORTRAN 4-49, 65
Resetting
COBOL 3-79
CYBIL 6-65
FORTRAN 4-70
Pascal 5-65
SCL 2-57
Showing
COBOL 3-87
CYBIL 6-72
FORTRAN 4-79
Pascal 5-73
SCL 2-62
Target 7-27, 31
Text object 1-6
Usage 1-1
Variables 2-34
Writing a definition 7-137
Form definition decks
COBOL 3-15
CYBIL 6-14
FORTRAN 4-14
Pascal 5-16
Form definition record
COBOL 3-4
Create from existing form 7-42
CYBIL 6-3

FORTRAN 4-3
Pascal 5-4
Format validation 7-6
FORTRAN
Call definitions G-1
Displaying forms 4-1
Program 4-15
Status checking 4-29
Subroutines 4-29
Validation formats 7-6
Full screen A-2
Full screen definition A-3
Function A-3
Function key assignments A-3
Function keys
Glossary definition A-3
See also Events

G

GET_FORM_VARIABLE 2-44
GETFV 2-44
Getting
Form
Attributes 7-56, 126
Names 7-127
Objects 7-128
Form definition record attributes 7-77
Help form attributes 7-57
Object attributes 7-81, 130
Record attributes 7-131
Stored object 7-132
Table attributes 7-75, 133
Variable
COBOL 3-45, 52, 58
CYBIL 6-38, 43, 47
FORTRAN 4-39, 46, 51
Pascal 5-40, 46, 49
SCL 2-34, 44
Variable attributes 7-69, 134
Getting a real variable
COBOL 3-52
CYBIL 6-43
FORTRAN 4-46
Pascal 5-46
SCL 2-34, 44
Getting a record
COBOL 3-55
CYBIL 6-45
FORTRAN 4-49
Getting a string variable
COBOL 3-58
CYBIL 6-47
FORTRAN 4-51
Pascal 5-49
SCL 2-34, 44
Getting an integer variable
COBOL 3-45
CYBIL 6-38
FORTRAN 4-39
Pascal 5-40
SCL 2-34, 44
Getting the next event
COBOL 3-48
CYBIL 6-40
FORTRAN 4-42
Pascal 5-42
SCL 2-64
Graphic object
Definition 1-6
Properties 7-9

H

Help
Creating 7-16
Default form 7-17
Defining the event 7-10
Displaying 7-13
Erasing 7-13
Form attribute 7-54, 57
Information 7-16
Hidden text 7-4
Highlighting
Display attribute 7-51
Setting attribute
COBOL 3-85
CYBIL 6-70
FORTRAN 4-77
Pascal 5-71
SCL 2-60
Validation 7-6
Hotline 11

I

Identifier A-3
Input
  COBOL 3-67
  CYBIL 6-55
  Format 7-61
  FORTRAN 4-60
  Pascal 5-57
  SCL 2-54
Instructions for
  Creating forms 7-22
  Using forms
    COBOL 3-1
    CYBIL 6-1
    FORTRAN 4-1
    Pascal 5-1
    SCL 2-1
Integer A-3
  Getting
    COBOL 3-45
    CYBIL 6-38
    FORTRAN 4-39
    Pascal 5-40
    SCL 2-34, 44
  Replacing
    COBOL 3-69
    CYBIL 6-57
    FORTRAN 4-61
    Pascal 5-59
    SCL 2-34, 55
Interactive mode A-3
Introduction 1-1

L

Language of form 7-55
Line drawing 7-2, 9
Line mode
  COBOL 3-84
  CYBIL 6-69
  FORTRAN 4-76
  Pascal 5-70
Local file A-3
Login A-4
Logout A-4

M

Main menu A-4
MANAGE_FORMS
  Command 2-46
  Functions 2-63
  Options 2-30
  Variable creation 2-31
Managing forms
  COBOL 3-1
  CYBIL 6-1
  Example 1-6
  FORTRAN 4-1
  Overview 1-2
  Pascal 5-1
  SCL 2-1
MANF 2-46
Mark display attribute
  Create 7-106
  Delete 7-119
Master catalog A-4
Menu
  See Events
Message
  Creating 7-10
  Form attribute 7-55, 58
Moving
  Objects 7-135
  Text 7-135
Multiple forms 7-9

N

Name A-4
Natural language
  COBOL 3-34
  CYBIL 6-29
  FORTRAN 4-28
  Pascal 5-31
  SCL 2-30
Normal task 7-11
NOS/VE A-4

O

Object attributes
  Changing 7-88
  Description 7-78
  Getting 7-130

Objects
  Copying 7-96
  Creating 7-99, 108
  Defining 7-3
  Deleting 7-118, 120
  Form 7-3
  Glossary definition A-4
  Moving 7-135
  Resetting attribute
    COBOL 3-80
    CYBIL 6-66
    FORTRAN 4-71
    Pascal 5-66
    SCL 2-60
  Setting attribute
    COBOL 3-85
    CYBIL 6-70
    FORTRAN 4-77
    Pascal 5-71
    SCL 2-60
Occurrence A-4
Online examples
  Accessing H-1
Online manuals
  Accessing B-1
  Glossary definition A-5
OPEF 2-48
OPEN_FORM 2-48
Opening a form
  COBOL 3-61
  CYBIL 6-49
  FORTRAN 4-54
  Pascal 5-51
  SCL 2-48
Operations
  See Events
Ordering printed manuals B-1
Output format 7-5, 63
Overview 1-1

P

Paging and scrolling 7-12
Parameter definitions
  Copying COBOL 3-3
  List of COBOL D-1
Pascal
  Displaying forms 5-1, 32
  Program 5-17
  Status checking 5-32
  Status constants E-1
  String convention 5-5
Permanent catalog A-5
Permanent file A-5
POP_FORM 2-49
POPF 2-49
Popping a form
  COBOL 3-63
  CYBIL 6-51
  FORTRAN 4-56
  Pascal 5-53
  SCL 2-49
POSITION_FORM 2-50
Position of cursor
  COBOL 3-82
  CYBIL 6-67
  FORTRAN 4-73
  Moving 7-14
  Pascal 5-68
  SCL 2-59
Position of event 2-66
Positioning a form
  COBOL 3-64
  CYBIL 6-52
  FORTRAN 4-57
  Pascal 5-54
  SCL 2-50
Procedure definitions
  Copying CYBIL 6-2
  Copying Pascal 5-2
Procedures
  Accessing 1-4
  Creating forms 7-85
  Displaying forms
    CYBIL 6-30
    Pascal 5-32
    SCL 2-36
  SCL 2-19
  Storing them 2-28
Processing events
  Abnormal
    COBOL 3-8
    CYBIL 6-7
    FORTRAN 4-7
    Pascal 5-9
    SCL 2-7

Normal
COBOL 3-7
CYBIL 6-6
FORTRAN 4-6
Pascal 5-8
SCL 2-6
Processor of form 7-55
Program A-5
Converting data 7-92, 94
Data type 7-66
Interaction with a form 7-9
Output 7-4
Record
COBOL 3-55, 75
CYBIL 6-45, 61
FORTRAN 4-49, 65
Tasks 7-11
Protected text 7-16; A-5
Prototype
COBOL 3-8
CYBIL 6-7
FORTRAN 4-7
Pascal 5-9
SCL 2-8
PUSF 2-52
PUSH_FORM 2-52
Pushing forms
COBOL 3-66
CYBIL 6-54
FORTRAN 4-59
Pascal 5-56
SCL 2-52

Q

QUI 2-53
QUIT 2-53

R

READ_FORM 2-54
Reading a form
COBOL 3-67
CYBIL 6-55
FORTRAN 4-60
Pascal 5-57
SCL 2-54
REAF 2-54
Real variable
Getting
COBOL 3-52
CYBIL 6-43
FORTRAN 4-46
Pascal 5-46
SCL 2-34, 44
Replacing
COBOL 3-72
CYBIL 6-59
FORTRAN 4-63
Pascal 5-61
SCL 2-34, 55
Record attributes
Getting 7-131
Record definition
Writing 7-138
Record form
COBOL 3-55, 75
CYBIL 6-45, 61
FORTRAN 4-49, 65
Rectangle form
Example
COBOL 3-11
CYBIL 6-10
FORTRAN 4-10
Pascal 5-12
SCL 2-13
Program 7-34
Related manuals B-1
REPFV 2-55
REPLACE_FORM_VARIABLE 2-55
Replacing a real variable
COBOL 3-72
CYBIL 6-59
FORTRAN 4-63
Pascal 5-61
SCL 2-34, 55
Replacing a record
COBOL 3-75
CYBIL 6-61
FORTRAN 4-65
SCL 2-34, 55
Replacing a string variable
COBOL 3-77
CYBIL 6-63
FORTRAN 4-67
Pascal 5-63
SCL 2-34, 55

Replacing an integer variable
  COBOL 3-69
  CYBIL 6-57
  FORTRAN 4-61
  Pascal 5-59
  SCL 2-34, 55
RESET_FORM 2-57
Resetting a form
  COBOL 3-79
  CYBIL 6-65
  FORTRAN 4-70
  Pascal 5-65
  SCL 2-57
Resetting an object attribute
  COBOL 3-80
  CYBIL 6-66
  FORTRAN 4-71
  Pascal 5-66
  SCL 2-60
RESF 2-57

S

SCL A-5
  Displaying forms 2-1, 36
  Procedure 2-22
Screen Design Facility
  Definition 1-4
  Usage 1-4
Screen Formatting
  Capabilities 1-12
  Definition 1-1
  Process 1-2
Screen updating
  COBOL 3-67, 87
  CYBIL 6-55, 72
  FORTRAN 4-60, 79
  Pascal 5-57, 73
  SCL 2-54, 62
Scrolling and paging 7-12
Select form example
  COBOL 3-10
  CYBIL 6-9
  FORTRAN 4-9
  Pascal 5-11
  SCL 2-12
SET_CURSOR_POSITION 2-59
SET_OBJECT_
 ATTRIBUTE 2-60
SETCP 2-59
SETOA 2-60
Setting an object attribute
  COBOL 3-85
  CYBIL 6-70
  FORTRAN 4-77
  Pascal 5-71
  SCL 2-60
Setting line mode
  COBOL 3-84
  CYBIL 6-69
  FORTRAN 4-76
  Pascal 5-70
Setting the cursor position
  COBOL 3-82
  CYBIL 6-67
  FORTRAN 4-73
  Pascal 5-68
  SCL 2-59
SHOF 2-62
SHOW_FORM 2-62
Showing a form
  COBOL 3-87
  CYBIL 6-72
  FORTRAN 4-79
  Pascal 5-73
  SCL 2-62
Size of table
  COBOL 3-38
  CYBIL 6-32
  FORTRAN 4-32
  Pascal 5-34
  SCL 2-38
Software support hotline 11
Special character A-5
Standard events 7-15
Standard function keys 7-45
Starting
  MANAGE_FORMS 2-3
  Prototype 2-8
Starting the application
  Commands to enter
    COBOL 3-34
    CYBIL 6-29
    FORTRAN 4-26
    Pascal 5-31
    SCL 2-30

Creating a user procedure
COBOL 3-32
CYBIL 6-27
FORTRAN 4-26
Pascal 5-29
SCL 2-28
Creating a user prolog
COBOL 3-32
CYBIL 6-28
FORTRAN 4-26
Pascal 5-30
SCL 2-29
Status checking
COBOL 3-35
FORTRAN 4-29
Pascal 5-32
Stopping MANAGE_FORMS 2-3
Stored object
Changing initial value 7-89
Creating 7-113
Deleting 7-121
Getting 7-132
String convention 5-5
String variable
COBOL 3-58, 77
CYBIL 6-47, 63
FORTRAN 4-51, 67
Pascal 5-49, 63
SCL 2-34, 44, 55
Submitting comments 11
Subroutines
Acessing 1-4
COBOL 3-35
FORTRAN 4-29
Symbol A-5
System Command Language A-5

T

Table
Attributes 7-73
Changing attributes 7-90
Creating 7-115
Deleting 7-122
Getting attributes 7-133
In a form 7-8
Object properties 7-8
Paging 7-12
Scrolling 7-12
Size
COBOL 3-38
CYBIL 6-32
FORTRAN 4-32
Pascal 5-34
SCL 2-38
Target form 7-27; A-6
Tasks for events 7-10
Temporary file A-6
Terminal
Definition keys C-2
Function keys 7-52
Input 7-4
Output 7-4
Session A-6
Update screen
COBOL 3-67, 87
CYBIL 6-55, 72
FORTRAN 4-60, 79
Pascal 5-57, 73
SCL 2-54, 62
User and program interaction 7-9
Text
Constant 7-3
Copying 7-96
Creating design 7-102
Deleting 7-118
Hidden 7-4
Moving 7-135
Properties 7-3
Protected A-5
Variable 7-3
Text object
Constant 7-3
Definition 1-6; 7-3
Table 7-8
Variable 1-7; 7-3
Transferring variables
COBOL 3-45, 52, 58, 69, 72, 77
CYBIL 6-38, 43, 47, 57, 59, 63
FORTRAN 4-37, 46, 51, 61, 63, 67
Pascal 5-40, 46, 49, 59, 61, 63
SCL 2-34

Types, CYBIL F-1

U

Unprotected text 7-16; A-5
User data 7-2, 92
User input
  COBOL 3-67
  CYBIL 6-55
  FORTRAN 4-60
  Pascal 5-57
  SCL 2-54
User name A-6
User prolog
  Creating
    COBOL 3-33
    CYBIL 6-28
    FORTRAN 4-27
    Pascal 5-30
    SCL 2-29

V

Validation
  Defining attributes 7-6
  Initial values 7-55
Variable
  Creating
    CYBIL 7-116
    SCL 2-31
  Deleting 7-123
  Getting
    COBOL 3-45, 52, 58
    CYBIL 6-38, 43, 47
    FORTRAN 4-39, 46, 51
    Pascal 5-40, 46, 49
    SCL 2-34, 44
  Replacing
    COBOL 3-69, 72, 77
    CYBIL 6-57, 59, 63
    FORTRAN 4-61, 63, 67
    Pascal 5-59, 61, 63
    SCL 2-34, 55
  Validation of initial values 7-55
Variable attributes
  Changing 7-91
  Creating 7-60
  Defining 7-3
  Getting 7-134
Variable text objects
  Definition 1-7
  Requirements 1-7
  Table 7-8
  Usage 7-3

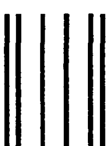
W

Wrap characters 7-5
Wrap words 7-5
Writing
  Form definition 7-137
  Program to use forms
    COBOL 3-2
    CYBIL 6-2
    FORTRAN 4-2
    Pascal 5-2
    SCL 2-3
  Record definition 7-138

Comments (continued from other side)

fold on dotted line;
lges with tape only.

FOLD



NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY MAIL
First-Class Mail Permit No. 8241 Minneapolis, MN

POSTAGE WILL BE PAID BY ADDRESSEE

CONTROL DATA
Technical Publications
ARH219
4201 N. Lexington Avenue
Arden Hills, MN 55126-9983

NOS/VE Screen Formatting 60488813 D

We would like your comments on this manual to help us improve it. Please take a few minutes to fill out this form.

Who are you?	**How do you use this manual?**
☐ Manager	☐ As an overview
☐ Systems analyst or programmer	☐ To learn the product or system
☐ Applications programmer	☐ For comprehensive reference
☐ Operator	☐ For quick look-up
☐ Other ______	☐ Other ______

What programming languages do you use? ______

How do you like this manual? Answer the questions that apply.

Yes	Somewhat	No	
☐	☐	☐	Does it tell you what you need to know about the topic?
☐	☐	☐	Is the technical information accurate?
☐	☐	☐	Is it easy to understand?
☐	☐	☐	Is the order of topics logical?
☐	☐	☐	Can you easily find what you want?
☐	☐	☐	Are there enough examples?
☐	☐	☐	Are the examples helpful? (☐ Too simple? ☐ Too complex?)
☐	☐	☐	Do the illustrations help you?
☐	☐	☐	Is the manual easy to read (print size, page layout, and so on)?
☐	☐	☐	Do you use this manual frequently?

Comments? If applicable, note page and paragraph. Use other side if needed.

Check here if you want a reply: ☐

Name	Company
Address	Date
	Phone

Please send program listing and output if applicable to your comment.

CONTROL DATA