

VOS COBOL Forms Management System

Notice

The information contained in this document is subject to change without notice.

STRATUS COMPUTER, INC., MAKES NO WARRANTY OF ANY KIND WITH REGARD TO THIS MATERIAL, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. Stratus Computer, Inc., shall not be liable for errors contained herein or for incidental or consequential damages in connection with the furnishing, performance, or use of this material.

Stratus Computer, Inc., assumes no responsibility for the use or reliability of its software on equipment that is not furnished by Stratus Computer, Inc.

This document is protected by copyright. All rights are reserved. No part of this document may be photocopied, reproduced, or translated to another language without the prior written consent of Stratus Computer, Inc.

Stratus, Continuous Processing, VOS, StrataNET, and the Stratus logo are registered trademarks of Stratus Computer, Inc.

XA, StrataLINK, and the SQL/2000 logo are trademarks of Stratus Computer, Inc.

Manual Name: *VOS COBOL Forms Management System*

Part Number: R035
Revision Number: 01
Printing Date: June 1989
VOS Release Number: 9.0

Stratus Computer, Inc.
55 Fairbanks Blvd.
Marlboro, Massachusetts 01752

©1985, 1989 by Stratus Computer, Inc. All rights reserved.

Contents

PART 1: Introduction to FMS

1. Overview	1-1
FMS Concepts and Terminology	1-1
Components of FMS	1-3
The Forms Editor	1-5
The screen Statements	1-6
The Forms Processor	1-8
2. Developing an FMS Application	2-1
The Development Process	2-1
Planning the Form	2-2
Information to Display to the User	2-2
Information to Be Entered by the User	2-3
Required Information and Optional Information	2-3
Arranging the Fields	2-3
Communication between the Form and the Program	2-4
Incidental Data	2-5
Traps	2-6
The Complete Form Layout	2-7
The Forms Editor	2-8
Invoking the Forms Editor	2-8
Creating Fields and Background Text	2-9
Setting the Form Options	2-18
The Set/modify form options Form	2-18
The MASKKEYS Form	2-19
Testing the Form	2-20
Changing Video Attributes	2-21
Fields	2-22
Required Fields	2-22
General Background Text	2-23
Specific Background Text	2-23
Saving the Form	2-24
Ending the Forms Editor Session	2-25
Writing the Application Program	2-25
Declaring the Variables and Constants	2-26
Field-Value Variables	2-26

Field-ID Constants	2-26
Other Variables	2-27
Initializing the Display List	2-27
Altering Initial Values	2-28
Displaying the Form	2-28
Processing the Form Data	2-29
The (ENTER) Key	2-30
Creating a Data-Entry Loop	2-31
Manipulating Data States	2-36
Changing a Display Type	2-39
Forms Input Mode	2-42
Compiling and Binding the Application	2-42

PART 2: FMS Reference Guide

3. The Elements of FMS	3-1
Form Components	3-1
Fields	3-1
Numeric and Alphanumeric Fields	3-2
Input Fields and Output Fields	3-3
Null Field Values	3-4
Initial Output Values	3-5
International Character Set Support	3-6
Specifying the Character Set for a Field	3-7
Storing ICSS Strings	3-7
3270 Device Dependencies	3-8
Forms and the Application Program	3-9
Operations on Forms	3-9
Defining a Form	3-9
Initializing a Form	3-10
Displaying a Form	3-10
Modifying a Form	3-11
Getting Information about a Form	3-11
Saving a Form	3-12
Discarding a Form	3-12
Control Transfer between an Application and a Form	3-13
Form Submission	3-13
Form Cancellation	3-14
Traps	3-14
Form Time-Out Period	3-15
Form Knocked Down	3-16
Output-Only Forms	3-16
Other Situations	3-16
4. The Forms Editor	4-1
Overview of the Forms Editor	4-1
The Form Design Process	4-2

The Two Forms Editors	4-2
The icss_edit_form Command	4-4
Definitions	4-10
Entering Text	4-11
Edit Requests	4-12
Direct Edit Requests	4-13
Menu Edit Requests	4-21
The Add/modify field Request	4-25
Modifying Existing Fields	4-26
Creating New Fields	4-26
Simple Fields	4-26
Array Fields	4-27
Uncommitted Fields	4-27
The Add/modify field Form	4-28
The Field Options	4-29
Display-Type Options	4-36
The Insert window field Request	4-41
The Set/modify form options Request	4-43
The Define/modify video display modes Request	4-51
The Files Produced by the Forms Editor	4-52
The Form Definition File	4-53
The Form Object Module	4-53
The Field-Values File	4-53
The Field-IDs File	4-54
The Forms Editor Keystrokes File	4-54
The Field Definition Files	4-54
Contents of the Form Definition File	4-56
The Field-IDs Include File	4-58
The Field-Values Include File	4-61
 5. Form Options	5-1
Form Option Summary	5-1
Form Option Reference Guide	5-2
 6. Field Descriptions	6-1
Defining a Field	6-2
Defining a Form Dynamically	6-2
Adding a Field Dynamically	6-4
Modifying a Field	6-6
Obtaining Field Information	6-6
Deleting a Field	6-7
Field Option Summary	6-8
Field Option Reference Guide	6-9
 7. Display Types	7-1
Display-Type Descriptions	7-2
Classes of Display Types	7-3

Predefined Display Types	7-4
Global Display Types	7-4
Temporary Display Types	7-5
Setting the Action and Visual Attributes	7-5
Cycle Display Types	7-7
Indexed Cycle Lists	7-9
Obtaining Cycle List Values	7-11
Display-Type Option Reference Guide	7-11
8. Data States	8-1
The Data-State Switches	8-1
Data States in the Forms Editor	8-2
Referencing Data States in a Program	8-2
Data-State Variables	8-2
Reading Data States	8-4
Single Field	8-4
All Fields	8-4
Changing Data States	8-5
Data State Reference Guide	8-6
9. Field Pictures and Filtering	9-1
Field Pictures	9-1
Specifying a Picture	9-2
Alphanumeric Pictures	9-3
Numeric Pictures	9-3
Field Precision	9-3
Periods and Commas	9-3
Filtering	9-4
Special Numeric Characters	9-5
The Filtering Process	9-6
Filtering Output Values	9-6
Filtering Input Values	9-7
10. Error Handling and Field Validation	10-1
Handling Forms Errors	10-1
Field Validation	10-4
The Validation Suite	10-4
Programmer-Defined Field Validation Routines	10-5
11. Windows and Subforms	11-1
Defining a Window Field	11-1
Initializing the Forms	11-2
Displaying the Forms	11-3
Example of Windows	11-4
12. Form Caching	12-1

The Forms Reference Count	12-2
Example of Form Caching	12-3
13. Traps	13-1
Establishing Traps	13-1
Trap on Field Entry	13-2
Trap on Field Exit	13-3
Returned Values	13-3
Field Validation	13-3
Example Using a Trap on Field Exit	13-4
Vertical Scroll Trap	13-5
Scrolling between Forms	13-5
Scrolling within a Form	13-6
Forms Input Mode	13-10
14. Subroutines	14-1
Forms Input Mode	14-1
s\$begin_forms_input	14-3
s\$end_forms_input	14-4
15. Built-In Functions	15-1
Built-In Function Summary	15-1
Built-In Function Reference Guide	15-4
16. Statements	16-1
The accept Statement	16-2
The perform screen delete Statement	16-6
The perform screen discard Statement	16-8
The perform screen initialization Statement	16-9
The perform screen input Statement	16-12
The perform screen inquire Statement	16-15
The perform screen output Statement	16-17
The perform screen save Statement	16-19
The perform screen update Statement	16-21
Appendix A: The accept Statement	A-1
The Display List	A-2
Initial Display and Redisplay	A-2
The into and update Form Options	A-4
Field Modes	A-5
Initializing Field Modes	A-6
Modifying Field Modes	A-6
Modes Example	A-7
Field-Value Justification	A-9
Obsolete Field Options	A-9
Field Output Values	A-10

Predefined Form with the into or update Form Option	A-11
Initial Display	A-11
Redisplay	A-11
Predefined Form without the into or update Form Option	A-11
Initial Display	A-12
Redisplay	A-12
No Predefined Form	A-12
Initial Display	A-13
Redisplay	A-13
Appendix B: The edit_form Command	B-1
The edit_form Command	B-2
The Add/modify field Request	B-7
The Set/modify form options Request	B-10
The Insert literal Request	B-11
Appendix C: Converting Old-Style Applications	C-1
Converting Predefined Forms	C-1
Updating the Application Program	C-2
The into and update Options	C-3
The form and formid Options	C-3
Obsolete Options	C-4
Dynamically Altering Forms	C-5
Output-Only Forms	C-11
Multiple Forms	C-11
Binding the New Application	C-11
Appendix D: Terminal Requirements	D-1
Input Requests	D-1
The Forms Editor	D-1
Menu Edit Requests	D-1
Direct Edit Requests	D-2
FMS Applications	D-3
Output Requests	D-4
The Forms Editor	D-4
FMS Applications	D-5
Attribute Requests	D-5
Appendix E: Form Storage Sizes	E-1
Appendix F: Global Control Operations	F-1
Forms Input Mode	F-1
Knocking Down Forms	F-4
Appendix G: Stratus Character Code Set	G-1

Glossary	H-1
Index	X-1

Figures

1-1. Sample FMS Form	1-2
1-2. Overview of FMS	1-5
2-1. Layout of Sample Form	2-7
2-2. Sample Application Program	2-33
4-1. Forms Editor Terminology	4-11
4-2. Insert Mode and Overlay Mode	4-12
4-3. The Forms Editor Request Menu	4-21
4-4. Form Displayed by the (MENU) F (Add/modify field) Request	4-28
4-5. Form Displayed by the (MENU) I (Insert window field) Request	4-42
4-6. Form Displayed by the (MENU) S (Set/modify form options) Request	4-43
4-7. The Forms Editor Mask-Keys Form	4-45
4-8. Form Displayed by the (MENU) V (Define/modify video display modes) Request	4-51
4-9. Sample Form Definition File	4-56
4-10. Generalized Field-IDs File	4-59
4-11. Generalized Field-IDs File with a Prefix Specified	4-60
10-1. Sample Validation Routine	10-6
11-1. Example Program Using Subforms	11-6
12-1. Example Program Using Subforms	12-4
13-1. Vertical Scroll Trap Example Program	13-7

Tables

3-1. The Numeric Picture Characters	3-2
3-2. The Alphanumeric Picture Characters	3-3
3-3. Sample Field Pictures and Corresponding Null Values	3-4
4-1. Explanation of Current Word	4-10
4-2. The Forms Editor Field Options	4-29
4-3. Data Types for Field-Values Variable Declarations	4-34
4-4. Auxiliary Information for Data Types	4-35
4-5. The Forms Editor Display-Type Options	4-36
4-6. Form and Field Options Affected by ALTERABLE BY ACCEPT	4-47
4-7. Files Created by the Forms Editor	4-52
5-1. The screen and accept Statement Form Options	5-2
5-2. Codes Returned by the keyed Form Option	5-9
5-3. Characters Used in the maskkeys Form Option	5-10
6-1. The accept and screen Statement Field Options	6-8
7-1. The Display-Type Options	7-2
7-2. The Display-Type Classes	7-3
8-1. The Data-State Switches	8-1
9-1. The Field Picture Characters	9-2
9-2. Examples of Filtering	9-6
12-1. Form Reference Count Manipulation	12-3
15-1. Display-Type Built-In Functions	15-2
15-2. Field Built-In Functions	15-3
16-1. The FMS Statements	16-1
A-1. The Mode Switches	A-5
C-1. Replacements for Old Form and Field Options	C-4
C-2. Replacements for Mode Switches	C-5
D-1. Alternatives for Menu Input Requests	D-2
D-2. Forms-Related Generic Input Requests	D-3
E-1. Storage Requirements of Form Components	E-1
G-1. Stratus Character Code Set	G-1

Introduction

The Purpose of This Manual

The *VOS COBOL Forms Management System (R035)* documents how to write application programs that use the Forms Management System (FMS) to perform I/O to the user's terminal display.

Audience

This manual is intended for application programmers who are experienced in COBOL. Before working with the *VOS COBOL Forms Management System (R035)*, you should be familiar with the *VOS COBOL Language Manual (R010)*.

Revision Information

This manual is a revision. For information on which release of the software this manual documents, see the Notice page.

The FMS program interface has been significantly redesigned since the last revision of this manual. The manual has been substantially rewritten to incorporate the new material and has been reorganized. Introductory material has been expanded to more fully describe the process of creating an FMS application.

Manual Organization

This manual has two parts, plus eight appendixes.

Part 1: "Introduction to FMS" consists of two chapters.

Chapter 1, "Overview," describes the major features and components of FMS.

Chapter 2, "Developing an FMS Application," describes in detail the process of creating a sample FMS application.

Part 2: "FMS Reference Guide" consists of 14 chapters.

Chapter 3, "The Elements of FMS," describes the basic components and concepts of FMS in a reference format.

Chapter 4, "The Forms Editor," provides the description of the `icss_edit_form` command and fully documents the features of the Forms Editor.

Chapter 5, "Form Options," describes each of the screen statement form options.

Chapter 6, “Field Descriptions,” discusses the field description clause of the screen statement and documents each of the field options.

Chapter 7, “Display Types,” explains how display types are manipulated and includes descriptions of each of the screen statement display-type options.

Chapter 8, “Data States,” describes how to read and update field data states within a program.

Chapter 9, “Field Pictures and Filtering,” discusses the use of field pictures to restrict field values and explains the process of filtering numeric data.

Chapter 10, “Error Handling and Field Validation,” explains how to handle status codes returned by screen statements and documents the field validation suite.

Chapter 11, “Windows and Subforms,” discusses the use of multiple windows to display more than one form on the screen at a time.

Chapter 12, “Form Caching,” describes how to save the internal image of a form so that it can be used later without being re-initialized.

Chapter 13, “Traps,” explains the use of features that can cause the form to be submitted when certain cursor movements occur.

Chapter 14, “Subroutines,” documents two subroutines related to FMS.

Chapter 15, “Built-In Functions,” documents a set of VOS COBOL functions related to FMS.

Chapter 16, “Statements,” gives the full syntax of the accept statement and each of the screen statements.

The first three appendixes document obsolete FMS features and describe how to convert old-style applications to use the newer features. Other appendixes describe the terminal features required to use FMS and the Forms Editor, discuss the storage size of forms, document some global control operations related to FMS, and provide tables of the ASCII and Latin alphabet No. 1 character sets.

Notation

Stratus documentation uses *italics* to introduce or define new terms. For example:

A field’s *null value* is the value displayed when the field is empty.

Computer font is used to represent text that would appear on your CRT screen or on a line printer. (Such text is referred to as *literal* text.) For example:

The picture `zzzzz9` limits each character in the field to a digit.

Slanted font is used to represent general terms that are to be replaced by literal values. In the following example, the term *form_name* shows that the user must supply an actual form name.

```
icss_edit_form form_name -into -cobol
```

Boldface is used to emphasize words within the text. For example:

The perform screen initialization statement does **not** display the form.

Syntax Notation

A *syntax format* is a specific arrangement of the elements of a VOS COBOL statement (or portion of a VOS COBOL statement). When VOS COBOL permits more than one arrangement, the documentation presents the arrangements as consecutively numbered formats. The text may also supply additional information defining the syntax formats.

The next table explains the syntax notation used to document the formats.

The Notation Used in Syntax Formats

Notation	Meaning
<i>element</i>	Required element.
<i>element</i> ...	Required element that can be repeated.
{ <i>element_1 element_2</i> }	Required list of elements.
{ <i>element_1 element_2</i> } ...	Required list of elements that can be repeated.
{ <i>element_1</i> <i>element_2</i> }	You are required to use one element of this set, but you cannot use more than one.
[<i>element</i>]	Optional element.
[<i>element</i>] ...	Optional element that can be repeated.
[<i>element_1 element_2</i>]	Optional list of elements.
[<i>element_1 element_2</i>] ...	Optional list of elements that can be repeated.
[<i>element_1</i> <i>element_2</i>]	Set of optional elements that are mutually exclusive; you can use only one.
Note: Dots, brackets, and braces are not literal characters; you should not type them. Any list or set of elements can contain more than two elements.	

In the table, the term *element* stands for the following elements of the VOS COBOL language:

- reserved words (in computer font)
- generic terms (in slanted font) that are to be replaced by such items as expressions, identifiers, literals, constants, or statements
- statements or portions of statements.

The elements in a list of elements must be entered in the order shown in a syntax format, unless the text specifies otherwise.

Brackets ([]) and braces ({ }) are sometimes nested in syntax formats.

Three dots (...) in a syntax format represents the position at which repetition can occur. The scope of the dots is either the preceding element or the segment of the syntax format enclosed in the preceding brackets or braces.

The following is an example of a syntax format used in this manual.

$$\text{cycle (} \left\{ \begin{array}{l} \text{cycle_value [,cycle_value] ...} \\ \text{cycle_value_array [,value_count]} \end{array} \right\} \text{)}$$

In examples, three dots in a column indicates that a portion of a language construct or program has been omitted. For example:

```
01  action_switches    comp-5.
01  visual_switches   comp-5.
   .
   .
   .
```

```
perform screen inquire with status (status_code),
       giving displaytype (12) action (action_switches)
       visual (visual_switches).
```

When an exact input or output value is shown, a special character, □, is sometimes used to represent a space character. For example, the following output value has three leading space characters:

□□□123

Format for Commands

Stratus manuals use the following format conventions for documenting commands.

command_name

The name of the command is at the top of the first page of the command description.

Privileged

This notation appears after the name of a command that can be issued only from a *privileged process*. (See the Glossary for the definition of privileged process.)

Purpose

Explains briefly what the command does.

CRT Form

Shows the form that is displayed when you type the command name and press the **DISPLAY FORM** key or when you type the command name followed by the option **-form**. The values displayed in the form are the command's *default values*. (See the Glossary for the definition of default value.)

The next table explains the notation used in CRT forms.

The Notation Used in CRT Forms

Notation	Meaning
■	Required field with no default value.
 	The cursor, which indicates the current position on the screen. It may be on the first character of a value, as in all .
<i>current_user</i> <i>current_module</i> <i>current_system</i> <i>current_disk</i>	The default value is the current user, module, system, or disk. The actual name will be displayed when you give the command.

Lineal Form

Shows the syntax of the command with its arguments. You can display an online version of the lineal form of a command by typing the command name followed by the option `-usage`.

The next table explains the notation used to document lineal forms. In the table, the term *multiple values* refers to explicitly stated separate values, such as two or more object names. Giving multiple values is **not** the same as giving a star name. (See the Glossary for the definition of star name.) When you give multiple values, you must separate the values with a space.

The Notation Used in Lineal Forms

Notation	Meaning
<code>argument_1</code>	Required argument.
<code>argument_1...</code>	Required argument for which you can give multiple values.
$\left\{ \begin{array}{l} \text{argument}_1 \\ \text{argument}_2 \end{array} \right\}$	You are required to give one argument of this set, but you cannot give both.
<code>[argument_1]</code>	Optional argument.
<code>[argument_1] ...</code>	Optional argument for which you can give multiple values.
$\left[\begin{array}{l} \text{argument}_1 \\ \text{argument}_2 \end{array} \right]$	Set of optional arguments that are mutually exclusive; you can give only one.
Note: Dots, brackets, and braces are not literal characters; you should not type them. Any list or set of arguments can contain more than two elements.	

Arguments

Describes the command's arguments. The next table explains the notation used to document command arguments.

The Notation Used in Argument Descriptions

Notation	Meaning
(CYCLE)	Only predefined values are allowed for this argument. In the CRT form, you display these values in sequence using the (CYCLE) key.
Required	You cannot enter the command without supplying a value for this argument. If an argument is required but has a default value, it is not labeled Required , since you do not have to include it when using the lineal form. With the CRT form, however, you must have a value in the field — either the displayed default or a value that you type in.
(Privileged)	Only a privileged process can give a value for this argument.

Explanation

Explains how to use the command and gives supplementary information. Not every command description requires an Explanation section.

Examples

Illustrates uses of the command.

Related Information

Refers you to related documentation (in this manual or other Stratus manuals), including descriptions of commands, subroutines, and requests you can use with or in place of this command.

Format for Subroutines

Stratus manuals use the following format conventions for documenting subroutines.

subroutine_name

The name of the subroutine is at the top of the first page of the subroutine description.

Purpose

Explains briefly what the subroutine does.

Usage

Shows how to declare the variables passed as arguments to the subroutine, declare the subroutine entry in a program, and call the subroutine.

Arguments

Describes the subroutine's arguments.

Explanation

Provides information about how to use the subroutine. Not every subroutine description requires an Explanation section.

Related Information

Refers you to other subroutines and commands similar to or useful with this subroutine.

Related Manuals

Refer to the following Stratus manuals for related documentation.

- *VOS COBOL Language Manual (R010)*
- *VOS COBOL Subroutines Manual (R019)*
- *VOS Communications Software: Asynchronous Communications (R025)*
- *VOS Communications Software: Defining a Terminal Type (R096)*

A Note on the Contents of Stratus Manuals

Stratus manuals document all the subroutines and commands of the user interface. Any other commands and subroutines contained in the operating system are intended solely for use by Stratus personnel and are subject to change without warning.

How to Comment on This Manual

You can comment on this manual by using the command `comment_on_manual`, described in the *VOS System Administrator's Guide (R012)*. Type `comment_on_manual`, press **(RETURN)**, and then complete the form that appears on your screen. You must fill in this manual's part number, R035. When you have completed the form, press **(ENTER)**. Your comments are sent to Stratus over the Remote Service Network. Note that the operating system includes your name with your comments.

Stratus welcomes any corrections and suggestions for improving this manual.

PART 1: Introduction to FMS

)

)

)

)

)

Chapter 1:

Overview

The Forms Management System (FMS) is a set of tools, system software, and programming language extensions that gives your programs a consistent interface for the entry and display of data on a video display terminal. The interface is designed to be efficient for applications in which the user performs transactions that each require the entry of several specific pieces of information.

FMS Concepts and Terminology

In FMS, the entire terminal screen, or a portion of it, is used as a *form* in which data can be entered or displayed. An FMS form is analogous to a paper form, such as a job application, tax return, or balance sheet. Just as a paper form allows for an orderly exchange of information (for example, between a job applicant and an employer; between a taxpayer and the Internal Revenue Service; between a corporation and its stockholders), an FMS form allows a program and a user to exchange information in an organized way.

An FMS form contains *fields* in which the program can display information or the user can enter information. The meaning of the data is derived from its location. For example, a specific field might be used for displaying or entering a telephone number. The same value entered in a different field might be interpreted as an invoice number. To indicate the field's meaning, a label might appear next to the field in the form. Labels and other text that appears outside of fields is called *background text*.

Figure 1-1 shows a sample FMS form.

Date: 09/26/89		
Customer name: _____	Customer number: _____	
New customer? no	Invoice number: 038A-10-45	
<u>Item number</u>	<u>Quantity</u>	<u>Price/unit</u>
_____	_____	_____
_____	_____	_____
_____	_____	_____
_____	_____	_____

Figure 1-1. Sample FMS Form

The form in Figure 1-1 contains fields in which the user can do the following:

- type a customer's name and number
- specify whether the customer is new
- enter several item numbers along with quantities and prices.

The values in the fields labeled `Date` and `Invoice number` are set by the program that displays the form and cannot be altered by the user. These two fields are *output fields*. The other fields are *input fields*.

In Figure 1-1, several of the input fields do not yet contain any data. These fields are said to be *null* or to contain their *null value*.

The user can move freely among the input fields of a form, typing and editing field values. Within each field, the user can issue most of the same editing requests as in a command line or in the CRT form of a command. Field values are not returned to the program until the user *submits* the form — for example, by pressing the **(ENTER)** key. When the form is submitted, control returns to the program. The user can also *cancel* the form, usually by pressing the **(CANCEL)** key. When the form is canceled, control returns to the program, but no field values are returned to the program.

Within the program, a specific variable receives the value from each field when the form is submitted. Such a variable is called a *field-value variable*.

The field labeled *New customer* in Figure 1-1 is a *cycle field*. A cycle field can accept only values from a specified list. The list of valid values is called the *cycle list*. In the case of the *New customer* field, the list might contain just two values: *yes* and *no*. The user cannot type a value in a cycle field. Instead, the user can change the field value by pressing either the **CYCLE** key or the left or right arrow key.

Note that fields in Figure 1-1 have different attributes and restrictions. Some fields appear in low intensity; others, in normal intensity. The field labeled *New customer* is a cycle field; the other fields are not. A set of specific characteristics such as these constitutes the field's *display type*. A display type determines the visual appearance of a field, specific actions associated with the field, and certain restrictions on field values. Display types provide a convenient means of specifying and dynamically changing field characteristics. Display types are fully described in Chapter 7, "Display Types."

Another set of field characteristics is collectively known as the field *data state*. A field data state is a series of switches that determine whether a field is input or output, whether it requires an input value or is optional, and so forth. These switches can be altered dynamically for each field. Other data-state switches return information about a field, such as whether the field value changed during the most recent form display. Data states are fully described in Chapter 8, "Data States."

Components of FMS

FMS has three major components:

- The Forms Editor, which allows you to create and modify forms.
- The screen statements. These statements are extensions to VOS COBOL that allow you to invoke a form within a program. (Another statement that invokes a form, the *accept* statement, is obsolete but is supported for existing applications.)
- The Forms Processor, the system software that is invoked by the screen statements. The Forms Processor manages form displays.

Each of these components is discussed later in this section.

In a typical forms application, the three components work together as follows:

1. Using the Forms Editor, you design a form. The Forms Editor stores the form description as a form object module that can be referenced by a program. (The Forms Editor also produces other files that are discussed later.)
2. Within a program, you reference the form in screen statements.

3. When you compile your program, the compiler translates the screen statements to object code that calls the Forms Processor run-time software.
4. When you bind your program, the binder links your program object module with the form object module and the Forms Processor software.
5. When control reaches a screen statement during program execution, the Forms Processor is invoked to manage the form. For example, when a perform screen input statement is executed, the Forms Processor displays the form. The Forms Processor then allows the user to move the cursor and type values within the form. When the user submits the form, the Forms Processor validates the input values and loads them into program variables specified within the screen statement. Control then returns to the program, and execution continues with the statement following the screen statement.

Figure 1-2 illustrates the FMS development process and execution.

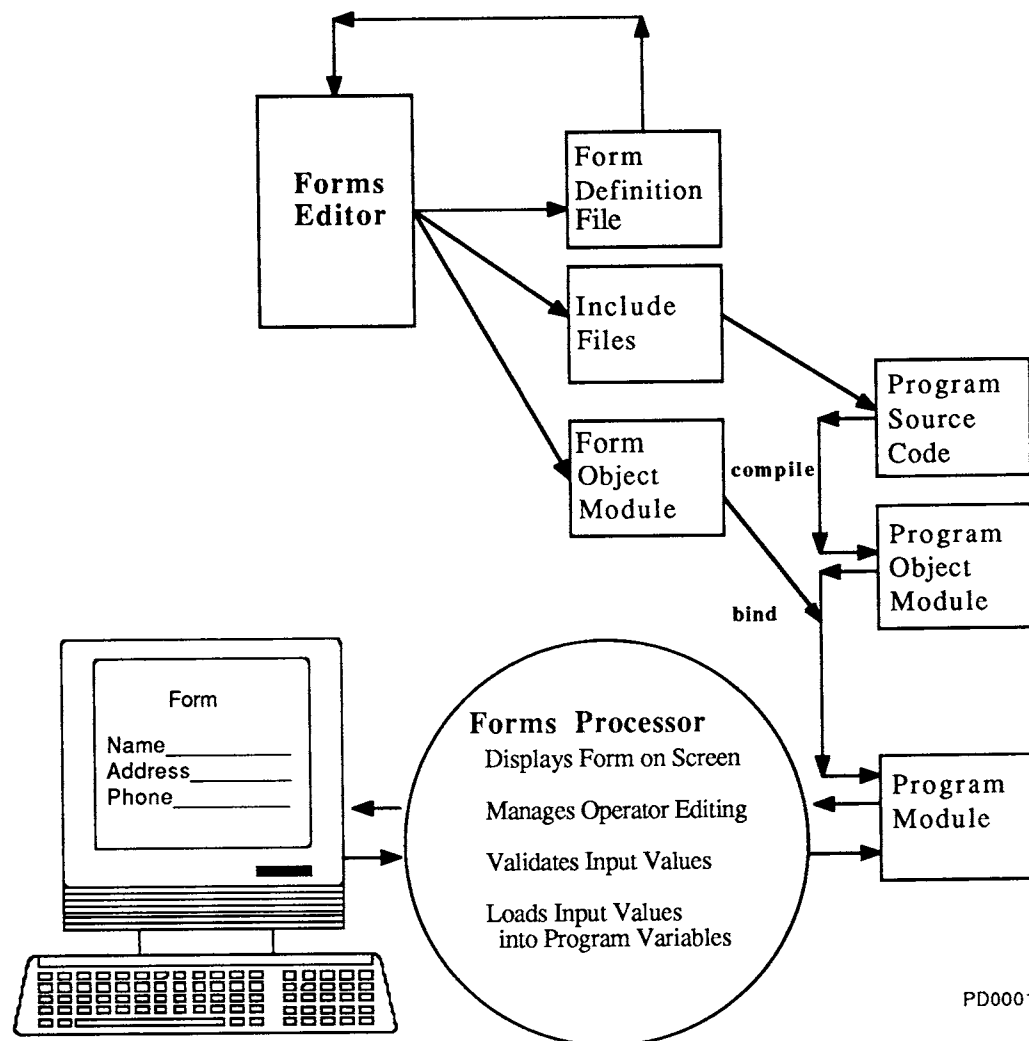


Figure 1-2. Overview of FMS

The Forms Editor

The Forms Editor is a program used to create, modify, and save FMS forms. To invoke the Forms Editor, issue the `icss_edit_form` command. (An older version of the Forms Editor, invoked by the `edit_form` command, is supported for older applications.)

The Forms Editor is similar to a word processor. It includes most of the direct edit requests found in the Word Processing Editor. In addition, the Forms Editor

contains a number of special menu requests. The menu requests allow you to do the following:

- add, delete, and modify fields in the form
- store field descriptions into a library and recall field descriptions from that library
- test the appearance and behavior of the form
- save the form description in two formats:
 - In a form definition file. This is a text file that you can modify in a subsequent invocation of the Forms Editor.
 - In a form object module. This file can be bound into a program.

Optionally, the Forms Editor generates the following include files for one or more programming languages:

- the *field-values file*, which contains declarations for the field-value variables that store the value of each field
- the *field-IDs file*, which defines mnemonic constants for the integer IDs that the Forms Processor uses to reference fields within the form.

A form created with the Forms Editor is called a *predefined form*. The fields defined with the Forms Editor are called *predefined fields*.

For more information on the Forms Editor, see Chapter 4, “The Forms Editor.” For information on the older version of the Forms Editor, see Appendix B, “The `edit_form` Command.”

The screen Statements

Eight screen statements have been added to VOS COBOL (the older `accept` statement is supported for older applications). The screen statements provide a forms interface for the COBOL programmer.

Briefly, the purpose of each screen statement is as follows:

- perform screen initialization creates an internal representation of a form in the user heap. This representation is called the *display list*. The perform screen initialization statement does not display the form.
- perform screen input displays a previously initialized form and accepts input from the user.
- perform screen output displays a previously initialized form but does not accept input from the user.
- perform screen update alters the display list but does not display the form.
- perform screen delete removes a field or a display type from the display list. The perform screen delete statement does not display the form.
- perform screen inquire returns information about the display list. The perform screen inquire statement does not display the form.
- perform screen save retains the storage of a form when that form is not currently displayed.
- perform screen discard cancels the effect of a perform screen save statement.

The first screen statement to operate on a form within a program must be the perform screen initialization statement. The other statements operate on the internal representation of the form that the perform screen initialization statement creates.

Usually, the perform screen initialization statement references a predefined form created by the Forms Editor. However, you can define a form dynamically within the perform screen initialization statement. Defining a form dynamically is difficult and inefficient; if possible, avoid doing so.

The next most important screen statement is the perform screen input statement. The perform screen input statement displays the form on the user's screen and allows the user to modify any input fields. Control does not return to the program until the user submits or cancels the form (unless the application uses special features).

The perform screen output statement displays the form on the user's screen, but does not allow the user to operate on the form in any way. Control returns immediately to the program.

The other screen statements modify the internal display list, but do not affect the user's screen. The user does not see changes made by these statements until a perform screen input or perform screen output statement is executed.

Each of the screen statements is described in Chapter 16, "Statements." The use of the perform screen save and perform screen discard statements is described in more detail in Chapter 12, "Form Caching."

The Forms Processor

The Forms Processor is the system run-time software that manages forms. When a screen statement is executed within a program, control transfers to the Forms Processor. The Forms Processor performs the operation indicated by the screen statement and then returns control to the program.

For example, when a perform screen initialization statement is executed, the Forms Processor reads the form definition and creates the display list. When a perform screen input statement is executed, the Forms Processor does the following:

- Reads the display list. (If the perform screen input statement contains options that update the display list, the Forms Processor performs those updates first.)
- Writes the form to the screen. (If the form has been displayed before, the Forms Processor determines whether the entire form must be rewritten, or if only certain fields must be updated.)
- Handles cursor movement and field editing requests from the user.
- Displays any help messages requested by the user.
- Validates field data when the user submits the form.
- Converts each field value to the data type of its field-value variable, and loads the converted values into the field-value variables.

When the Forms Processor completes the requested action, control returns to the application program. Execution continues with the statement following the screen statement.

Chapter 2:

Developing an FMS Application

An FMS application consists of one or more forms and an application program that invokes those forms. Developing an FMS application, therefore, involves designing the form or forms and writing the application program.

This chapter explains these two tasks by showing how to develop a sample form and how to write the program that invokes it. The sample application allows the user to enter data into an employee database.

The Development Process

The general procedure for developing an FMS application is as follows:

1. Plan the form.
2. Create the form with the Forms Editor.
3. Write the application program.
4. Compile and bind the application.

Note: It is possible to design a form from within the application program. However, when possible, it is usually easier and is always more efficient to define the form separately with the Forms Editor. You should define forms within the program only when the information needed to structure the form is not available until run time. For information on designing a form within the program, see the description of the perform screen initialization statement in Chapter 16, "Statements." (For applications using the accept statement, see Appendix A, "The accept Statement.")

In this chapter, the steps are arranged to clarify the decisions you must make and the operations you must perform to create an FMS application. In practice, you do not have to follow the procedures exactly as outlined in this chapter. For example, many programmers find it easier to plan the form while using the Forms Editor. The Forms Editor is designed to allow you to experiment with the form design.

Some programmers find it easier to write the program first and create the form afterwards. Use whatever procedure works best for you.

Note that the last step in developing an FMS application is always to compile the program and bind the application.

Planning the Form

This section describes the process of planning the form. The process can be summarized as follows:

1. Decide what information must be displayed to the user.
2. Decide what information can be entered by the user.
3. Decide which information is required and which is optional.
4. Design an appropriate layout of the information.
5. Plan the communication between the form and the program.

The following subsections explain each step in detail.

Information to Display to the User

Some forms serve primarily to display information to the user; others serve primarily as a means for the user to enter information into the program. The sample form developed in this chapter is of the latter type. However, two kinds of information must still be displayed to the user:

- Background text to explain the form fields and to help the user to fill in the appropriate information. The exact background text is determined when the form is designed. It cannot be changed by the program.
- Output information derived by the program. Some information to be displayed on the form cannot be determined until run time. For example, the form might display the current date, or it might display current information derived from a database.

The sample application will supply one piece of output information that is determined at run time: a unique ID number for each new employee. This will require an output field in the form. The form will also require background text to label the form input fields. The exact background text will be determined when designing the form layout.

Information to Be Entered by the User

The user will be allowed to enter the following information about each employee.

- Name (first, middle initial, and last)
- Social Security number
- Department number
- Salary

Each of these pieces of information requires an input field in the form. Because the name will be stored as three separate pieces of information (first name, middle initial, and last name) in the database, the form will use three input fields for that information.

Required Information and Optional Information

The user is required to enter some data, but other data is optional. In the sample application, the information entered by the user will be entered as a record in a database. The database requires the following information for each record.

- Employee ID number
- First name
- Last name
- Social Security number

The other items are optional. The first of the required pieces of information, the employee ID number, is generated by the program and displayed in an output field. The other required items are entered by the user. These fields will be *required* fields. This means that the user cannot submit the form until information is provided for these fields. All other input fields in the form will be *optional*. This means that the user does not have to supply values for these fields.

Arranging the Fields

Once you know the fields that the form must contain, you can begin to plan an appropriate arrangement of the fields. First, you must consider the overall size of the form.

Some forms fill the entire terminal display. The size of these forms is limited by the size of the screen on which they will be displayed. For example, the Stratus V101 and V102 terminals have 24 usable lines and 80 usable columns.

If a form is to be displayed within a window in another form, or if it must share the screen with other information, additional constraints are imposed on form size.

The sample form is a full-screen form. It will be designed to fit within 24 lines and 80 columns.

At least one space character must immediately precede and follow each field.

Designing an appropriate arrangement for fields within a form is largely subjective. However, the following guidelines can be helpful.

- Group related information together.
- Put the most important or distinctive information near the top of the form.
- Arrange the information in a format that is convenient or familiar to the user.
- Leave space around information to improve readability. Once you have determined the size constraints of the form, use the available space.

For the sample application, the most distinctive piece of information is probably the employee's name. Therefore, the employee's name should appear at the top of the form. The fields will be arranged approximately as follows:

Name: _____
(first) (middle) (last)

Employee number: #### Social Security number:

Department number:

Salary: \$ _____ per month

Note that labels are used to the left of the fields. In some cases, additional labels below fields clarify what information is expected from the user. In addition, the salary field is preceded by a dollar sign and followed by the words “per month” to clarify what value is expected. More detailed information will be available to the user through the (HELP) request. The help text for the sample form is established within the Forms Editor.

Communication between the Form and the Program

When you have planned the design of the form, you must determine when and how control will transfer between the form and the application program.

Early in the form design process you have to begin thinking about why the program invokes the form. This allows you to decide what information the form must display and what information the user should enter. More complex questions about the interaction between the form and the program arise later in the design process.

If a form is used strictly to display information to the user, communication between the form and the program is almost always simple. If the program is obtaining input through the form, communication can be simple or complex.

Incidental Data. In a simple case, the program displays the form, the user fills in all the appropriate information and submits the form, and the program receives the input. Note that the input received by the program can include all of the following:

- the data in each field when the user submitted the form
- what action caused the form to be submitted (for example, which key the user pressed to submit the form)
- the location of the cursor when the form was submitted.

Each of these pieces of information can be meaningful. For example, the key used to submit the form might indicate what is to be done with the field values. One key might indicate that the program should add the values to a database, and another key might indicate that the program should search a database for matching values. For more information on using the submission key to indicate information, see the description of the `MASKKEYS` option under the heading “The Set/modify form options Request” in Chapter 4, “The Forms Editor,” and the description of the `maskkeys` form option in Chapter 5, “Form Options.”

Similarly, the cursor location might indicate the action to be taken by the program. For example, the form might contain several fields specifying different actions such as depositing or withdrawing money, or checking an account balance. The user could indicate which action the program should perform by positioning to the corresponding field and submitting the form. For more information on using the cursor position to indicate information, see the description of the `getcursor` option in Chapter 5, “Form Options.”

If an application uses indicators such as the key used or the cursor position as data, the meaning of that data must be understood by the user filling in the form and by the application program that processes the form. When you design the form, you should include background text to explain the meaning of such indicators to the user.

In the sample application, the key used by the user will determine what the program does with the data. The user can indicate either that the data be added directly to the employee database, or that the data be held elsewhere until approved. The application activates two function keys for this purpose.

Note: The forms software allows you to enable VOS generic function keys. The specific keys to which the generic function keys are mapped depend on the terminal-type definition. For the sample application, assume that it will be run on a V102 terminal.

To make the meanings of the function keys known to the user, add the following to the bottom of the form:

FUNCT-1: Add to employee database

FUNCT-3: Hold for approval

Traps. In some cases, one or more fields must be filled in by the user before the program can determine all the values that it must display in the form. To handle this, the application can display the form and allow the user to fill in only some of the fields before *trapping*, or returning to the program. The program can then determine additional output values and redisplay the form with those values. The user can then continue filling in field values and finally submit the form.

Traps can be associated with individual fields. You can establish a trap that occurs whenever the user moves the cursor into a specific field. You can also establish a trap that occurs whenever the user moves the cursor out of a specific field. For further information on these kinds of traps, see the descriptions of the TRAP ON FIELD ENTRY and TRAP ON FIELD EXIT options under the heading "The Add/modify field Request" in Chapter 4, "The Forms Editor." See also the descriptions of the TRAP_ON_FIELD_EXIT and TRAP_ON_FIELD_ENTRY switches in the discussion of the action display-type option in Chapter 7, "Display Types."

The sample application does not require any traps.

The Complete Form Layout

Figure 2-1 shows the entire layout of the form. Note that the label **Employee Information** has been added to identify the form. For a more complex form, you can give a label to each section of the form.

Employee Information

Name: _____
 (first) (middle) (last)

Employee number: #### Social Security number: _____

Department number: ____

Salary: \$ _____ per month

FUNCT-1: Add to employee database

FUNCT-3: Hold for approval

Figure 2-1. Layout of Sample Form

The Forms Editor

This section describes how to use the Forms Editor to create a form. The main steps in the process are as follows:

1. Invoke the Forms Editor.
2. Create the fields and background text.
3. Set the form options.
4. Test the form.
5. Save the form.
6. End the Forms Editor session.

Each of these steps is explained in detail in the following subsections. An additional subsection describes how to change video attributes within a form.

Invoking the Forms Editor

Invoke the Forms Editor with the `icss_edit_form` command. This command is fully described in Chapter 4, “The Forms Editor.” For the sample application, only four of the command arguments are of interest.

- `input_path`
- `form_path`
- `-into`
- `-cobol`

The `input_path` and `form_path` arguments indicate the form definition files to be accessed by the Forms Editor. The `input_path` argument specifies a file from which an existing form definition is to be read. The `form_path` argument specifies the file to which the new or modified form definition is to be written. Both the `input_path` and `form_path` file names must end with the suffix `.form`, but you can omit that suffix in the command line.

By default, the name of the form is the simple name of the output file, `form_path`, without the suffix `.form`. For example, if the name of the output file is `%s1#d01>Sales>Jones>menu.form`, the default form name is `menu`.

The `input_path` argument is required. The `form_path` argument is optional. If you omit the `form_path` argument, the file you specify in `input_path` is used for both reading and writing.

For the sample application, a new form is being created; no existing form definition is used. However, the `input_path` argument is still required. Since you do not want to modify an existing form, you must supply the name of a form definition file that does not exist. You could give a random string of characters; but remember that the file name you specify is also the default value for `form_path`. If you give the name you want for the output file in `input_path`, you do not need to specify a value for `form_path`. Also, this is a way of checking that the file name you choose is not

already in use. If a form definition file with that name exists, the Forms Editor displays an edit buffer containing information on the existing form definition.

The `-into` and `-cobol` arguments indicate what include files the Forms Editor is to generate when it writes the form definition. The language arguments, such as `-cobol`, indicate for which languages the Forms Editor is to generate field-IDs files. If you specify `-into`, then the Forms Editor also creates a field-values file for each programming language you specify. For more information on these include files, see Chapter 4, "The Forms Editor."

For the sample application, the name of the new form will be `employee_info`. The application program that invokes the form will be written in VOS COBOL and will require a field-values file. Assume that the current directory does not currently contain a file named `employee_info.form`. The following command invokes the Forms Editor:

```
icss_edit_form employee_info -into -cobol
```

When you issue this command, the Forms Editor is invoked. It searches for a file named `employee_info.form` in the current directory, but does not find it. It then displays an empty edit buffer and puts you at edit request level.

Creating Fields and Background Text

When the Forms Editor first displays the edit buffer, the cursor is in the upper left-hand corner of the buffer. You can move the cursor and type background text in the same way that you would in the Word Processing Editor. For more information, see "Direct Edit Requests" in Chapter 4, "The Forms Editor."

For example, to put the title `Employee Information` in the middle of the fourth line, do the following:

1. Move the cursor down three lines by pressing the down-arrow key, **[↓]**, three times.
2. Move the cursor to the right by pressing the **[TAB]** key five times.
3. Type the heading `Employee Information`.

Next, move the cursor down three more lines. Position to the beginning of the line by issuing the **[GOTO]** **[←]** request. Type a space character, and then type the label of the first field, `Name:`. Note that you must leave the first character position of each line blank.

Type another space character after the label. The cursor is now positioned to where the first field will begin. Note that you must leave a space character immediately to the left of each field.

To add a field in the current cursor location, issue the `Add/modify field` request. This is a Forms Editor menu request. To issue a menu request, you first press the **[MENU]** key. The Forms Editor then displays a screen of requests. Each request

has a letter associated with it. Find the request you want to issue, and type the associated letter. The letter for the Add/modify field request is F. You can issue the request by typing an uppercase or lowercase F while the request menu is displayed. The Add/modify field request is commonly called the (MENU) F request.

When you issue the (MENU) F request, the Forms Editor displays a screen in which you can enter information about the field you are defining. In this example, the screen will appear as follows:

Field Options			
FIELD NAME	name		
POSITION	7 , 8		
ARRAY LAYOUT		ROW SPACING	COLUMN SPACING 1
LENGTH			
FIELD TYPE	input		
DISABLE	no		
REQUIRED	no		
DISAPPEARING	no		
IN FIELD-VALUES	yes		
SHIFT, DCS =	none	field-values sequence	0
INITIAL			
HELP			
Displaytype Options			
DISPLAYTYPE NAME			
PICTURE			
VALUE RESTRICTION	none	VALIDATE	
INTENSITY	high	underline	not inverse
		non blinking	not blanked
JUSTIFICATION		TRIM BLANKS	yes
AUTO TAB	no	BANK TELLER DECIMAL	no
TRAP ON FIELD EXIT	no	TRAP ON FIELD ENTRY	no
FORCE INSERT MODE	no	FORCE OVERLAY MODE	no

Note that the Forms Editor is itself displaying an FMS form. For complete information on the (MENU) F form, see Chapter 4, "The Forms Editor."

The Forms Editor derives the initial value of FIELD NAME from the background text that appears to the left of the field. In the current example, this field name is not appropriate. The form being created has three name fields: first name, middle initial, and last name. The field currently being defined is for the first name. Therefore, change the value to first_name. This change does not change the background text or the form's appearance.

The Forms Editor derives the POSITION values from the current cursor location. In the current example, the cursor was located at row 7, column 8. If you change

these values, the location of the field changes. In this example, the values do not need to be changed.

All other values displayed in the (MENU) F form are default values. You can change the characteristics of the field by changing these values.

Before submitting the (MENU) F form, you must indicate the length of the field you are defining. You can do this in either of two ways: you can specify a value for the LENGTH field option, or you can specify a value for the PICTURE display-type option. For the first_name field, specify 24 for LENGTH.

Recall that first_name is a required field. To enforce this, set the REQUIRED field to yes. The REQUIRED field is a cycle field. To change the value, position to the field and press the (CYCLE) key.

It is a good practice to provide a help message for every input field in a form. The user can display the help message to get information to aid in filling in the field. To set a help message for the first_name field, position to the HELP field and type the following message:

Enter the employee's first name.

When you have set all the information appropriately, submit the (MENU) F form by pressing the (ENTER) key. Rather than returning you to the edit buffer, the Forms Editor redisplay the (MENU) F form that you have just filled in. The following message appears at the bottom of the screen:

Data type information has changed.

Two new fields appear to the right of the FIELD TYPE field. The new fields are labeled COBOL. The first field contains the value display-2 and the second field contains the value pic x(48). These fields indicate the data type to be used in the field-values file for the variable associated with the first_name field. The Forms Editor derives the default data type from the length of the field and the field picture (if any). In this example, the default COBOL data type is pic x(48) display-2. For information on how the Forms Editor derives data types, see Chapter 4, "The Forms Editor." You can change the data type by cycling the first data-type field and by overwriting the extent value in the second data-type field. In the current example, use the data type derived by the Forms Editor.

The screen should now appear as follows:

Field Options			
FIELD NAME	first_name		
POSITION	7	,	8
ARRAY LAYOUT			ROW SPACING COLUMN SPACING 1
LENGTH	24		
FIELD TYPE	input	COBOL display-2	pic x(48)
DISABLE	no		
REQUIRED	yes		
DISAPPEARING	no		
IN FIELD-VALUES	yes		
SHIFT, DCS =	none	field-values sequence	0
INITIAL			
HELP	Enter the employee's first name.		
Displaytype Options			
DISPLAYTYPE NAME			
PICTURE			
VALUE RESTRICTION	none	VALIDATE	
INTENSITY	high	underline	not inverse
		non blinking	not blanked
JUSTIFICATION		TRIM BLANKS	yes
AUTO TAB	no	BANK TELLER DECIMAL	no
TRAP ON FIELD EXIT	no	TRAP ON FIELD ENTRY	no
FORCE INSERT MODE	no	FORCE OVERLAY MODE	no

Resubmit the (MENU) F form by again pressing the (ENTER) key. This time, the submission should succeed. The Forms Editor returns you to the edit buffer. A row of 24 block characters now appears in the buffer to hold the place of the first_name field. The cursor is immediately to the left of the block characters.

If you want to change the values you specified in the (MENU) F form, leave the cursor positioned immediately to the left of the field (or move it back to there) and reissue the (MENU) F request. The same form is redisplayed with the values you specified. You can then modify any value you wish and resubmit the (MENU) F form.

When you have finished with the first_name field, move the cursor to the right by pressing the (→) key. Note that the cursor jumps from the beginning of the field to the end of the field. Press the (→) key again, and the cursor jumps two more positions to the right. This is the first position in which you can create a new field in the current row. Recall that each field must be preceded and followed by a space.

Create the middle_initial field at the current location. To do this, you must issue the (MENU) F request and again fill in the form displayed by the Forms Editor. In this case, the FIELD NAME field is initially blank because no background text immediately precedes the field. Type middle_initial into the FIELD NAME field.

To establish the length of the field, you can set the `LENGTH` field to 1, or you can set the `PICTURE` option to `U`. The picture `U` restricts the field to a single uppercase letter. For information on field pictures, see Chapter 9, "Field Pictures and Filtering."

Do not change the setting of the `REQUIRED` field. The middle initial is not required information.

Set the `HELP` field to the following:

Enter the employee's middle initial (if any).

As an added convenience to the user, cycle the `AUTO TAB` field to `yes`. This means that as soon as the user enters a value that fills the `middle_initial` field, the cursor moves automatically to the next field. The user does not have to manually move the cursor. This option is useful if the field value has a known length and the user can be expected to normally move immediately to the next field after completing this field. For the `middle_initial` field, the length of the value (if given) is always one character, and the user will normally want to fill in the last name field immediately after the middle initial.

If you were to set the `AUTO TAB` field to `yes` for a field that accepts values of different lengths, the behavior of the field might seem inconsistent to the user. If the user types a value that fills in the entire field, the cursor automatically moves to the next field, but in other cases, the cursor does not move automatically. In these cases, it is usually best not to use the `AUTO TAB` feature.

When you first submit the `(MENU) F` form for the `middle_initial` field, the form is redisplayed with the data-type fields added. The default data type is `pic x(2) display-2`. You do not need to change this type. Resubmit the form to return to the edit buffer. The `middle_initial` field is represented by a single block character.

Move the cursor eight positions to the right of the `middle_initial` field, and create the `last_name` field. Within the `(MENU) F` form, set the values as follows:

<u>FIELD NAME</u>	<u>LENGTH</u>	<u>REQUIRED</u>	<u>AUTO TAB</u>
<code>last_name</code>	<code>24</code>	<code>yes</code>	<code>no</code>

Also, compose an appropriate help message.

When you submit the form, the Forms Editor redisplay the form with the data-type fields indicating the `pic x(48) display-2` type. You do not need to change this value. Resubmit the form.

Next, move to the line below the name fields, and type in the following background text directly below the corresponding fields. Refer to Figure 2-1 for the exact location.

(first) (middle) (last)

Move down three more lines and position the cursor to the left of the screen. Type a space character, and then type the background label for the next field: Employee Number:. Type a space character, and then issue the (MENU) F request.

Unlike all the preceding fields, the `employee_number` field is output-only. Recall that the employee number will be generated by the application and displayed to the user. Therefore, cycle the FIELD TYPE field to output only. Fill in the rest of the (MENU) F options as follows:

<u>FIELD NAME</u>	<u>LENGTH</u>	<u>REQUIRED</u>	<u>AUTO TAB</u>	<u>PICTURE</u>
<code>employee_number</code>	6	no	no	zzzzz9

The picture `zzzzz9` limits each character in the field to a digit. The field characters represented by `z` picture characters will be suppressed if they are leading zeros. This means, for example, that the value 32 will appear in the field as 32, rather than as 000032. The single 9 character ensures that the value zero is represented as 0, rather than as a blank field. For information on pictures, see Chapter 9, "Field Pictures and Filtering."

When you first submit the (MENU) F form, not only do the data-type fields appear, but several of the display-type field values change. The following message appears at the bottom of the screen:

The modes have changed (according to the rules for default modes).

The INTENSITY field changes from high to low, and the value underline changes to no underline. These are the default values for output-only fields. These values appear when you define an output-only field without changing the standard defaults. You could change the values back to the previous setting (or to any other setting) if you wish. For the current example, leave the values set to the defaults for output-only fields.

The data type derived by the Forms Editor for the `employee_number` field is `pic x(12) display-2`. However, since the field value is strictly numeric, it is better to store it in a numeric variable. Therefore, cycle the first data-type field to `comp-5`, and delete the value 12 from the second data-type field. This changes the field data type to `comp-5`. Resubmit the form to return to the edit buffer.

The remaining fields in the form are all input fields. Type the background text for the `social_security_number` field and create that field. Refer to Figure 2-1 for the appropriate layout. Within the (MENU) F form, enter the following values:

<u>FIELD NAME</u>	<u>LENGTH</u>	<u>REQUIRED</u>	<u>AUTO TAB</u>	<u>PICTURE</u>
<code>social_security_number</code>	11	yes	yes	999-99-9999

Compose an appropriate help message for the field.

Accept the data type derived by the Forms Editor. Resubmit the form to return to the edit buffer.

Move down three more lines, and type the background text for the department_number field. Type a space and issue the (MENU) F request. The department_number field differs from other fields in that it is a cycle field. This means that the allowed values for the field are restricted to a specific list of values. Rather than typing a value in the field, the user cycles to the correct value using the (CYCLE), (CYCLE BACK), and arrow keys.

Within the (MENU) F form enter the following values:

<u>FIELD NAME</u>	<u>LENGTH</u>	<u>REQUIRED</u>	<u>AUTO TAB</u>	<u>PICTURE</u>
department_number	3	no	no	999

In addition, cycle the VALUE RESTRICTION display-type option to cycle. This establishes that department_number is a cycle field.

Also, compose an appropriate help message.

When you submit the (MENU) F form, not only do the data-type fields appear, but the value of the underline mode changes to no underline. The following message appears at the bottom of the screen:

The modes have changed (according to the rules for default modes).

By default, cycle fields are not underlined. If you want the field to be underlined, you can change the mode value before resubmitting the form. For the current example, accept both the default modes and the data type derived by the Forms Editor.

When you submit the (MENU) F form for the second time, rather than returning you to the edit buffer, the Forms Editor displays a form in which you can enter cycle values. This form appears as follows:

Enter a list of cycle values.

0	___
1	___
2	___
3	___
4	___
5	___
6	___
7	___
8	___
9	___
10	___
11	___
12	___
13	___
14	___
15	___
16	___
17	___
18	___
19	___

You can enter up to 20 cycle values for the field. Each cycle value is three characters wide because you specified 3 as the length of the field. The allowable values for the department_number field are 010, 020, 030, and 040. Remember, however, that the department_number field is not required. To allow the user to omit this field, include a blank value in the cycle list.

The first value in the cycle list is the default initial output value for this field. You can override this default by specifying a different value in the INITIAL field of the (MENU) F form. You can also override it from within the application program. For the current example, the field should be initially blank. Therefore, leave the first value in the cycle list blank, and put the other department_number values in fields 2 through 5 of the cycle-list form. Blank fields at the end of this form are ignored, so in this example, the cycle list contains five values.

After typing the values, press (ENTER) to submit the cycle-list form. The Forms Editor returns you to the edit buffer. If you subsequently wish to alter the cycle list, position to the department_number field and issue the (MENU) F request. The Forms Editor displays the (MENU) F form. When you submit that form, the Forms Editor displays the cycle-list form again with the values you specified previously. You can then update the list and resubmit the form to return to the edit buffer.

Move the cursor down three more lines in the edit buffer, and type the label that precedes the salary field as in Figure 2-1. Note that the label includes a dollar sign (\$) to clarify the expected field value. Type a space after the dollar sign, and issue the (MENU) F request. The initial value for FIELD NAME is salary_\$. Change this value to salary. Set the other values in the (MENU) F form as follows:

FIELD NAME	LENGTH	REQUIRED	AUTO TAB	PICTURE
salary	9	no	no	zz,zz9.99

Also, provide an appropriate help message.

Submit the form. The data type derived by the Forms Editor is pic x(18) display-2. Because the value of salary is a number that might be used in arithmetic operations, it is more efficient to store it as numeric data. Therefore, cycle the first data-type field to comp-6 and change the second data-type field to pic s9(7)v9(2). This establishes pic s9(7)v9(2) comp-6 as the data type of the field.

After resubmitting the (MENU) F form and returning to the edit buffer, type the background text per month after the field. This further clarifies the value the user should type in the field. Note that you must leave a space between the end of the field and the background text.

Type the rest of the form background text as shown in Figure 2-1. This additional text indicates the meaning of function keys to the user.

Note: You might find that the form is too long to fit in the edit window. However, if you try to move beyond the last line of the window, the displayed text scrolls up and you can complete the form. If you then move the cursor up beyond the beginning of the edit window, the top part of the form is again displayed. The final form must fit on the terminal screen on which it is displayed, but the edit window is shorter than the screen. You can check the size of the form by using the (MENU) X request, Show exact form, as described later in this section. You can also use the (MENU) N request, En/disable line number mode, to display line numbers in the edit buffer. The (MENU) N request is discussed in Chapter 4, "The Forms Editor."

This completes the definition of fields and background text for the form. The next step is to set some general form attributes.

Setting the Form Options

This subsection describes how to set some general attributes and options for the form.

The Set/modify form options Form. Issue the **(MENU) S** request, Set/modify form options. When you issue this request, the Forms Editor displays the following form.

```

-- Form Options -- for employee_info

PREFIX
MASKKEYS      no
PRODUCE INTO   yes
BASIC         no
COBOL         yes
FORTRAN       no  STRINGS yes
PASCAL        no
PL/1         no
C            no
VALIDATE      and report errors
WIDE CURSOR   no
MESSAGE
CURSOR FIELD  _____ INDEX  1
ERROR MESSAGE FIELD

BACKGROUND MODE
  INTENSITY:      low      no underline      not inverse
                  non blinking      not blanked

REQ FIELD MODE TOGGLES
  INTENSITY TOGGLE: same intensity  same underlining  toggle inverse
                  same blinking     same blanking

```

The **(MENU) S** form is described fully in Chapter 4, "The Forms Editor."

The cursor appears initially in the MASKKEYS field.

Note that the form name, `employee_info`, appears at the top of the form. This is the form name derived by the Forms Editor based on the name of the output file. You can position to the form name and change its value. For the sample application, leave the default form name.

The values of the `PRODUCE INTO` and `COBOL` fields are yes because you specified the `-into` and `-cobol` options on the command line. The other language fields are initialized to no because you did not specify the corresponding command-line options. As discussed earlier in this chapter, these options determine what include files are created when you save the form. You can override the command-line

options by changing these values in the (MENU) s form. For the sample application, leave these fields as they are.

You can position to the PREFIX field and type a prefix for the form. The prefix you specify is applied to field-ID names in the field-IDs file. If an application uses more than one form, you should specify a unique prefix for each form to ensure that no two fields have the same field-ID name. Since the sample application uses only one form, a prefix is not necessary.

Because the form uses function keys, you must cycle the MASKKEYS field to yes. If MASKKEYS is yes, then when you submit the (MENU) s form, the Forms Editor displays another form in which you can specify what function keys to enable.

Leave the other fields in the (MENU) s form set to the default values. For information on these fields, see Chapter 4, "The Forms Editor."

Make sure the value of MASKKEYS is yes and submit the (MENU) s form. Because MASKKEYS is set to yes, the Forms Editor displays the MASKKEYS form.

The MASKKEYS Form. The following is the MASKKEYS form displayed by the Forms Editor.

Define Key Mask																															
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32
B	B	B	B	B	B	B	B	B	B	B	B	B	B	B	B	B	B	B	B	B	B	B	B	B	B	B	B	B	B	B	B
E <=> "enter", B <=> "beep" and C <=> "cancel"																															

This form allows you to associate an action with each of the 32 VOS generic function keys.

Most terminal types map specific terminal keys to some (but not all) of the generic function keys. For example, the standard VOS v102 terminal type maps the first generic function key to the sequence (FUNCT) (1). The user issues this sequence by

holding down the **(FUNCT)** key and pressing **(0)**. The first 19 generic function keys have similar mappings in the v102 terminal type. For information on key mappings for a specific terminal, ask your system administrator, or refer to the terminal type definition file in the *VOS Communications Software: Defining a Terminal Type (R096)*.

The MASKKEYS form contains a cycle field for each of the 32 generic function keys. The value for each field indicates what the Forms Processor should do if the user issues a key sequence associated with that generic function key from within the form. The meanings are as follows:

- E Submit the form.
- B Beep and take no further action.
- c Cancel the form.

The default value for each key, B, indicates that the Forms Processor should beep and take no other action if the user issues that key from within the form. The user remains within the form until the form is either submitted or canceled.

For the sample application, the form should be submitted if the user issues the sequence associated with either the first or third generic function key. Therefore, cycle the first and third fields in the form to E. Leave the other fields set to the default value, B.

Submit the MASKKEYS form and return to the edit buffer. If you wish to modify the values you specified in the MASKKEYS form, reissue the **(MENU)** s request and resubmit that form with the MASKKEYS option set to yes. The Forms Editor again displays the MASKKEYS form. You can change any of the displayed values and resubmit the form.

You have now completed the definition of the employee_info form. Before saving the form and ending the edit session, you should test the form as described in the next subsection.

Testing the Form

The **(MENU)** x request, Show exact form, allows you to see how the form you have defined will appear to a user. Issue this request. The Forms Editor clears your screen and displays the employee_info field that you have defined. The first_name, last_name and social_security_number fields appear in inverse video because they are required. Most of the other fields contain a number indicating the field ID.

You can attempt to submit the form by pressing the **(ENTER)** key or by issuing the sequence associated with either the first or third generic function key. However, if you have not supplied a value for each of the required fields, the submission fails.

Fill out the form in the same way that a user might. Use a typical value for each field. Note that you cannot position to the employee_number field because it is output-only.

Within each field, you can press the **(HELP)** key to display the help text for that field.

Cycle through the possible values for the `department_number` field.

When you position to the `salary` field, the cursor is initially located at the decimal point. Type a number of dollars, and then type a decimal point. (Do **not** type a dollar sign.) When you type the decimal point, the cursor moves to the right. You can now type a number of cents.

You can now submit the form with either the **(ENTER)** key or one of the enabled function keys, or you can cancel the form by pressing the **(CANCEL)** key.

If you submit the form, the Forms Processor checks the values you have specified. If the form passes validation, the Forms Editor returns you to the edit buffer. If you cancel the form, the Forms Editor returns you to the edit buffer without validating any field values.

You might want to repeat the test of the form for various field values. Try submitting the form with each of the enabled function keys.

As you test the form, consider the appearance of the screen. You might decide, for example, that the descriptions of the function keys should stand out from the rest of the background text. One way to make them stand out better is to make them visually different than the other background text. The following subsection describes how to change the video attributes of text within a form.

When you have finished testing the form, return from the test buffer to the normal edit buffer by submitting or canceling the form.

Changing Video Attributes

Each field and each segment of background text in a form has certain video characteristics. The following are the five video attributes supported by FMS:

- low, normal, or high intensity
- blinking or not blinking
- underlined or not underlined
- inverse video or not inverse video
- blanked or not blanked.

The Forms Editor allows you to change the following video attributes:

- the video attributes of each field
- the video attributes used to designate required fields
- the video attributes of all background text
- the video attributes of specific background text.

Note: Video attributes are highly device-dependent. Your terminal might not support all of the attributes supported by FMS.

Fields. To change the video attributes of a specific field, first position to that field and invoke the `(MENU) F` form. Within the display-type options in that form are five cycle fields that specify the field's video attributes. You can position to each of these fields and cycle to a new value. When you have set the attributes as you want them, submit the `(MENU)` F form to return to the edit buffer.

In the current example, leave the field video attributes set to their default values.

Required Fields. When the Forms Processor displays a form to the user, it changes one or more video attributes of each required field that is initially null. This indicates that the user must supply a non-null value for the field. By default, the Forms Processor changes the inverse attribute of a field to indicate that the field is required. That is, if the field is normally displayed in non-inverse video, it is displayed in inverse video if it is required; if the field is normally displayed in inverse, it is displayed in non-inverse. The other video attributes of the field are unchanged. As soon as the user types in the field, the field reverts to its normal attributes.

You can make the Forms Processor use any video attribute, or any combination of video attributes, to indicate required fields. The Forms Editor `(MENU) S` form contains a section labeled `REQ FIELD MODE TOGGLES`. The five fields in this section of the form indicate how required fields with null values are to be displayed. For each of the five attributes, you can do the following:

- specify that the attribute be the same as it normally is for the field (for example, same blinking)
- specify that the attribute be switched, or toggled, to the opposite of the normal value for the field (for example, toggle blinking)
- specify a specific setting for the attribute regardless of the normal setting for the field (for example, not blinking or blinking).

Note that the default is `toggle inverse`, with the other video attributes retaining their normal settings. For the sample form, use the defaults.

General Background Text. The **(MENU)** s form also contains five fields labeled BACKGROUND MODE. These five fields indicate the default video attributes of all background text in the form. The default settings are low intensity, no underline, not inverse, not blinking, and not blanked. For the sample form, use the defaults.

Note that these switch settings control the attributes of all background text in the form, unless you specify other attributes for specific areas of background text.

Specific Background Text. To set the attributes of a specific area of background text, do the following:

1. Set a mark at the beginning of the region.
2. Move the cursor to the end of the region. This causes the Forms Editor to highlight all text in the region.
3. Issue the **(MENU)** V request, Define/modify video display modes.

In the sample application, the descriptions of the function keys will stand out if they are in normal intensity and inverse video. Position the cursor to the F in FUNCT-1. Press the **(MARK)** key. Move the cursor to the right. Note that when you first move the cursor, text after the cursor moves right. Text between the mark and the cursor is highlighted. Move the cursor to the space following the word database. Issue the **(MENU)** V request, Define/modify video display modes. The Forms Editor displays the following form.

-- Video Display Modes --			
INTENSITY:	same intensity	same underlining same blinking	same inversion same blanking

The **(MENU)** V form has five fields that allow you to set five video attributes of the region of background text you have highlighted. For each attribute, you can specify one of the following:

- that the Forms Processor display the highlighted region with the same attribute as other background text (for example, same underlining)
- that the Forms Processor switch, or toggle, the attribute for the region to the opposite of the default value for background text (for example, toggle underlining)
- that the Forms Processor display the region with a specific setting for the attribute (for example, underlined or not underlined).

For the current region, set the intensity field to normal intensity, and set the inverse field to inverse. Submit the form to return to the edit buffer.

When you return to the buffer, the region that had been highlighted is no longer highlighted, but is underlined. The Forms Editor uses underlining within the edit buffer to indicate a region that has special video attributes. When you test the form with the (MENU) X request, the region has the attributes you specified.

Highlight the instruction that reads **FUNCT-3: Hold for approval**. Issue the (MENU) V request, and set the attributes for that region. When you are done, test the form with the (MENU) X request to see if the appearance is as expected.

Saving the Form

When you are satisfied with the appearance and behavior of the form, you can save the form definition by instructing the Forms Editor to write it to the form definition file. The path name of the file to which the Forms Editor writes the definition is determined when you issue the `icss_edit_form` command as described earlier in this chapter.

To save the form definition, issue the (MENU) W request, **Write**. The Forms Editor then writes the form definition to the output file. The Forms Editor also writes any include files that you have requested through command line options or (MENU) S options. Finally, the Forms Editor creates a form object module. The name of the object module is the same as the name of the form definition output file, except that the suffix `.form` is replaced by the suffix `.obj`.

For the sample form, the Forms Editor creates the file `employee_info.form` and writes the form definition to that file. It also creates the include files `employee_info_ids.incl.cobol` and `employee_info.incl.cobol`. Finally, it creates an object module named `employee_info.obj`.

When the (MENU) W request finishes, the terminal bell sounds and the following message appears at the bottom of your screen:

Include file(s) are new: You must recompile the programs which use them.

This message indicates either that the include files did not exist before or that they did exist before, but have now been modified. When creating a new form, you can ignore this message.

Ending the Forms Editor Session

After saving the form, you can end the Forms Editor session by issuing the **(MENU) Q** request, Quit.

If you have not saved the form before you issue the **(MENU) Q** request, or if you have modified the form since you saved it, the Forms Editor writes the following message:

A buffer has been modified and not written.

The Forms Editor then writes the following prompt:

Quit anyway?

If you answer yes or y, the Forms Editor ends the session and returns you to VOS command level. Any modifications you made since saving the form are lost. If you give any other answer, the Forms Editor returns you to the edit buffer. You can then save the form with the **(MENU) W** request and reissue the **(MENU) Q** request. The Forms Editor then ends the session and returns you to VOS command level.

At this point, the definition of the form is complete. The rest of this chapter deals with the other half of the forms application: the program that invokes the form.

Writing the Application Program

This section discusses a VOS COBOL application program that invokes a form. Only those parts of the program directly related to the Forms Management System are discussed. For general information on VOS COBOL, refer to the *VOS COBOL Language Manual (R010)*.

The following are the major steps in a typical FMS program.

1. Declare the variables and constants.
2. Initialize the display list.
3. Display the form and accept input through the form.
4. Process the returned data.

The following subsections discuss each of the steps in detail. Subsequent subsections discuss the following topics:

- creating a program loop to process multiple input records
- reading and changing the data state of a field
- changing the display type of a field
- using forms input mode.

Declaring the Variables and Constants

This subsection describes how to declare field-value variables and field-ID constants in a VOS COBOL program.

Field-Value Variables. In order to read information from a form field, you must associate a program variable with that field. This is also the most convenient way to write information to a field. Such variables are called *field-value variables*.

The data type of the field-value variable should be the same as the data type you specified in the COBOL field of the Forms Editor (MENU) F form. You can declare a record containing all the field-value variables for a form by using the field-values include file produced by the Forms Editor.

The field-values include file produced for the sample form, `employee_info.incl.cobol`, begins with a comment indicating when the form was last modified. The rest of the file is as follows:

```
02 first_name pic x(48) display-2.           // 07, 08
02 middle_initial pic x(2) display-2.        // 07, 35
02 last_name pic x(48) display-2.            // 07, 46
02 employee_number comp-5.                   // 11, 19
02 social_security_number pic x(22) display-2. // 11, 59
02 department_number pic x(6) display-2.     // 14, 45
02 salary pic s9(7)v9(2) comp-6.             // 17, 33
```

You can use this file to declare a record of field-value variables as follows:

```
01 employee_fields.
   copy 'employee_info.incl.cobol'.
```

For information on the copy statement, see the *VOS COBOL Language Manual (R010)*.

Field-ID Constants. To specify the cursor position within the form, or to change the data state or display type of a field, you must reference fields by their integer IDs. The field-IDs include file created by the Forms Editor specifies mnemonic names for all the field-IDs of a form.

Like the field-values file, the field-IDs file for the sample form, `employee_info_ids.incl.cobol`, begins with a comment indicating when the form was last saved. The rest of the file is as follows:

```
%replace first_name_id          by 1 // 07, 08
%replace middle_initial_id      by 2 // 07, 35
%replace last_name_id           by 3 // 07, 46
%replace employee_number_id     by 4 // 11, 19
%replace social_security_number_id by 5 // 11, 59
%replace department_number_id   by 6 // 14, 45
%replace salary_id              by 7 // 17, 33
%replace employee_info_max_ids  by 7
```

You can include this file in your program as follows:

```
copy 'employee_info_ids.incl.cobol'.
```

Other Variables. The sample application also requires the following declarations:

```
01 beep_switch          comp-4.
01 employee_form_id     comp-4.
01 error_code           comp-4.
01 key_code             comp-4.
```

The purpose of each of these variables is discussed in the following subsections.

Initializing the Display List

Before you can display a form, you must load that form into your address space. The loaded version of the form is called the *display list*. You invoke the Forms Processor to load the form (and initialize the display list) by executing the `perform screen initialization` statement.

The `perform screen initialization` statement is fully described in Chapter 16, “Statements.”

The sample application requires only the following simple version of the statement:

```
perform screen initialization 'employee_info'
      into (employee_fields)
      with formid (employee_form_id) status (error_code).
```

This version of the `perform screen initialization` statement has four operands. The value `'employee_info'` provides input from the program to the Forms Processor. The other operands return output from the Forms Processor. The first two operands, `'employee_info'` and the `into` option, are the *form specifier*. The value `'employee_info'` specifies the predefined form that is to be loaded. The `into` operand,

`employee_fields`, is a record that returns the initial value for each field. Note that this is the same record that you declared with the field-value include file. The `formid` operand, `employee_form_id`, returns an integer ID for the form. This form ID is used to reference the form in subsequent statements. The `status` operand, `error_code`, returns a VOS status code. If no error or exceptional condition occurs, the Forms Processor returns the value 0 in `error_code`.

After the perform screen initialization statement, the program should check the value of `error_code`. If it is not 0, the program should handle the error. For more information, see Chapter 10, "Error Handling and Field Validation."

Altering Initial Values

The perform screen initialization statement sets each member of the `employee_fields` record to the initial value for the corresponding field. For information on how initial field values are determined, see Chapter 3, "The Elements of FMS."

For the sample application, most of the returned initial values are acceptable. The one exception is the value for the `employee_number` field. Recall that this is an output-only field in which the application displays a unique ID number for the employee. Before the form is displayed, the application program must generate this unique value. The procedure by which this is done is not directly related to the forms software and is therefore beyond the scope of the current discussion. However, the application must assign an appropriate value to `employee_number` of `employee_fields` before displaying the form.

Displaying the Form

After the form is initialized, you can execute the perform screen input statement. This statement invokes the Forms Processor to display the form and allow the user to enter data. (Another statement, perform screen output, displays the form but does not accept input from the user.) The perform screen input statement is fully described in Chapter 16, "Statements."

For the sample application, the perform screen input statement is as follows:

```
perform screen input 'employee_info' update (employee_fields)
    with formid (employee_form_id) keyused (key_code)
    status (error_code).
```

This version of the perform screen input statement has four operands. The first operand provides input to the Forms Processor. The update option provides input and also returns output. The other operands are output-only.

The first operand, 'employee_info', indicates which form is to be displayed.

The update operand, `employee_fields`, is a record. On input, this operand specifies initial values to be displayed in the fields of the form. Recall that this record was

initialized in the perform screen initialization statement. On output, if the user submits the form, `employee_fields` returns the values that appeared in the fields of the form when it was submitted. If the user cancels the form, the values returned in `employee_fields` are unchanged.

The keyed operand, `key_code`, returns an integer code either indicating that the user canceled the form, or indicating which key the user pressed to submit the form. For the sample application, the values in the following table are possible.

Value	Meaning
-1	User canceled the form.
0	User submitted the form with the ENTER key.
1	User submitted the form with the sequence that maps to function-key-1.
3	User submitted the form with the sequence that maps to function-key-3.

For the complete list of values that `keyused` can return, see the description of the `keyused` option in Chapter 5, "Form Options."

The status operand, `error_code`, returns a VOS status code. You should check this value immediately after the perform screen input statement. If the returned value is not 0, you should handle the error appropriately. For more information, see Chapter 10, "Error Handling and Field Validation."

Processing the Form Data

After checking the returned status code, you can process the other data returned to the perform screen input statement. In the sample application, you should first check the value returned in the `keyused` option. The value should be interpreted as follows:

- If the returned value is -1, the user canceled the form. No new data is returned in the update option.
- If the returned value is 1, the user requests that the information returned in the update option be added directly to the employee database.
- If the returned value is 3, the user requests that the information returned in the update option be held for approval. (Add the data to a specific database for this purpose.)
- If any other value is returned in `keyused`, then the appropriate action is undefined. Handle this case as an error.

You can handle these actions through a series of nested if statements. For the current application, you can code a series of if statements as follows:

```
if key_code equal -1 then
    go to exit-program
else if key_code equal 1 then
    perform add-employee-record
else if key_code equal 3 then
    perform hold-employee-record
else go to fatal-error.
```

Note: To make the code more readable, you could create mnemonic constants for the expected key_code values. The following %replace statements create such constants for the sample application.

```
%replace ADD_EMPLOYEE by 1
%replace .CANCEL by -1
%replace HOLD_EMPLOYEE by 3
```

The ENTER Key

The ENTER key is always activated as a submission key; it cannot be disabled. An application should either assign a meaning to that key (and make that meaning known to the user), or specifically code for the case where the ENTER key is used.

If the form is submitted with the ENTER key, the value 0 is returned in keyused.

In the sample application, the **ENTER** key could have been used in place of, or as an alternative to, the first function key. Alternately, the application can effectively disable the key. That is, if the user submits the form with the **ENTER** key, the application could redisplay the form with the field values unchanged. This can be done by enclosing the perform screen initialization statement in a loop as follows:

```
%replace ENTER      by 0
.
.
.

move ENTER to key_code.
move 0 to error_code.
move 0 to beep_switch.

perform get-input until ((key_code not equal ENTER) or
                        (error_code not equal 0)).
.
.
.

get-input.

perform screen input 'employee_info' update (employee_fields)
      with formid (employee_form_id) beep (beep_switch)
      keyused (key_code) status (error_code).

move 1 to beep_switch.
```

In this example, whenever the user submits the form with the **ENTER** key, the program loops back and re-executes the perform screen input statement. This means that the form remains on the user's screen and the cursor is again positioned within the form. The beep option is used to sound the terminal bell on each form display after the first (that is, the bell sounds each time the user presses the **ENTER** key).

For more information on the beep option, see Chapter 5, "Form Options."

Creating a Data-Entry Loop

The application program described thus far in this chapter accepts data for only one employee. To enter data about a second employee, the user would have to re-invoke the program. The application could be more useful if it allowed the user to enter information about several employees in one invocation. You can do this by creating a data-entry loop within the program.

The application needs to execute the perform screen initialization statement only once. Therefore, this statement is not part of the loop.

The loop must do the following:

1. Make any necessary preparations or alterations for the fresh form display.
2. Display the form and accept input from the user.
3. Process that input.

Steps 2 and 3 are no different than before. (However, if you expect to process many records, you might prefer to just add them to a list in step 3 and process them fully after the data-entry loop.)

Step 1 involves the following items:

- setting (or resetting) the input fields to their initial values
- generating a unique value for the `employee_number` field
- positioning the cursor to the first field of the form.

Previously, the first of these items was performed by the `perform screen initialization` statement. Now, however, the field values must be reset for each pass through the loop. Executing a `perform screen initialization` statement for each pass is time-consuming. Instead, you can save a copy of the initial values before entering the loop. This implies that the application requires two sets of field-value variables: one set to hold the initial field values and another to hold the current field values. Therefore, in addition to the `employee_fields` structure, declare the following:

```
01 initial_fields.  
   copy 'employee_info.incl.cobol'.
```

To put the initial field values into this record, reference it instead of the `employee_info` record in the `perform screen initialization` statement. Within the data-entry loop, you must copy the values from `initial_fields` to `employee_info` before each fresh display of the form (including the first display).

After copying the initial values to `employee_info`, the application must set `employee_number` of `employee_info` to a unique value for each pass through the loop.

Finally, before displaying the form, the application should position the cursor to the first input field, `first_name`. On the first display of the form, this is done automatically, unless you specify otherwise. On subsequent displays, the cursor appears in the same field to which it was positioned when the form was last submitted, unless you specify otherwise. You can specify the initial cursor position for each display with the `putcursor` option of the `perform screen input` statement. You reference the position by field ID. Recall that the field-IDs file establishes mnemonic constants for field IDs. The constant for the `first_name` field is `first_name_id`.

Figure 2-2 shows the entire application program with the data-entry loop.

```

identification division.
program-id. enter_employees.

data division.
working-storage section.

    %replace ADD_EMPLOYEE          by 1
    %replace CANCEL                by -1
    %replace ENTER                 by 0
    %replace HOLD_EMPLOYEE        by 3

    copy 'employee_info_ids'.

01 employee_fields.
    copy 'employee_info'.

01 initial_fields.
    copy 'employee_info'.

01 beep_switch          comp-4.
01 cursor_field        comp-4.
01 employee_form_id    comp-4.
01 error_code          comp-4.
01 key_code            comp-4.

procedure division.

*   Initialize the display list.

    perform screen initialization 'employee_info'
        into (initial_fields)
        with formid (employee_form_id) status (error_code).

    if error_code not equal 0 then
        go to fatal-error.

```

Figure 2-2. Sample Application Program

(Continued on next page)

Figure 2-2. (Continued)

```
*   Execute the data entry loop.

    perform input-loop until error_code not equal 0.

    if error_code not equal 0 then
        go to fatal-error.

    go to exit-program.

input-loop.

*   Copy the initial values to the employee_fields structure.

    move initial_fields to employee_fields.

*   Move a new employee number to employee_number of employee_fields.
    .
    .
    .

*   Set (or reset) the initial cursor position to first_name field.

    move first_name_id to cursor_field.

*   Display the form until the user cancels or presses an
*   appropriate function key (reject ENTER).

    move ENTER to key_code.
    move 0 to error_code.
    move 0 to beep_switch.

    perform get-input until ((key_code not equal ENTER) or
                            (error_code not equal 0)).

    if error_code not equal 0 then
        go to fatal-error.
```

(Continued on next page)

Figure 2-2. (Continued)

```

*   Execute the action requested by the user.

    if key_code equal CANCEL then
        go to exit-program.

    if key_code equal ADD_EMPLOYEE then
        perform add-employee-record
    else if key_code equal HOLD_EMPLOYEE then
        perform hold-employee-record
    else go to fatal-error.

get-input.

*   Display the form.

    perform screen input 'employee_info' update (employee_fields)
        with beep (beep_switch) keyused (key_code)
        putcursor (cursor_field) status (error_code).

    move 1 to beep_switch.

add-employee-record.
    .
    .
    .

hold-employee-record.
    .
    .
    .

fatal-error.

*   Handle error.
    .
    .
    .

exit-program.

    exit program.

```

Note that a new variable, `cursor_field`, has been added to hold the field ID of the first field. The paragraphs `add-employee-record`, `hold-employee-record`, and `fatal-error` have been added, but left undefined. For information on error handling, see Chapter 10, "Error Handling and Field Validation."

Manipulating Data States

The sample application developed in this chapter does not read or alter the data state of any field. An application can read data states to determine whether a value has been specified for a field or whether the field value has been changed by the user. By altering a field's data states, you can change characteristics such as whether the field is required or optional, or whether the field is an input field or an output field.

For example, in the sample application, the `department_number` field is optional. By reading the data state of that field after the `perform screen input` statement, you can determine whether the user specified a value for that field or not. By changing the data state value, you can dynamically change `department_number` to a required field.

Within a screen statement, you can reference the data state of a specific field with the `datastate` field option, or you can reference the data states for all fields in the form with the `datastates` form option. This section describes the use of the `datastates` form option. For more information on the `datastate` field option, see Chapter 6, "Field Descriptions."

If you want to reference data states in a program, first declare a table to hold the data-state value of each field. The table must contain one element for each field. The field-IDs include file defines a constant, `form_name_max_ids`, that indicates the number of fields in the form. Each element of the table must be a `comp-4` item. Each element holds the binary coding of several switches. The VOS include file `(master_disk)>system>include_library>form_datastate.incl.cobol` defines mnemonic names for these switches as follows:

```
%replace FILTER_FOR_CONVERSION by 256
%replace DISABLE_ENTIRE_FIELD by 128
%replace NEW_DATA_IN_FIELD by 64
%replace INPUT_FIELD by 32
%replace REQUIRED_FIELD by 16
%replace MARKED_FIELD by 8
%replace FIELD_HAS_CHANGED by 4
%replace FIELD_VALUE_GIVEN by 2
%replace DISAPPEARING_DEFAULT by 1
```

The meaning of each data-state switch is described in Chapter 8, "Data States."

You can include the `form_datastate.incl.cobol` file in the sample program and declare a data-states table as follows:

```
copy 'form_datastate.incl.cobol'.

01 emp_data_state.
   20 data_state_value      comp-4 occurs employee_info_max_ids.
```

Note that this example assumes you have included the `field-IDs` file that contains the definition of `employee_info_max_ids`.

To initialize the `emp_data_state` table, use the `datastates` form option in the `perform screen initialization` statement.

```
perform screen initialization 'employee_info'
       into (initial_fields)
       with formid (employee_form_id) datastates (emp_data_state)
       status (error_code).
```

You can then reference the data state of an individual field, such as `department_number`, by subscripting the `emp_data_state` array with the field ID.

The data state for the `department_number` field is stored in `data_state_value (department_number_id)`. If you want to check if the `department_number` field is required, examine the `REQUIRED_FIELD` bit as follows:

```
01 quotient                comp-4.
   .
   .
   .

if data_state_value (department_number_id) >= 0 then
   compute quotient =
      data_state_value (department_number_id) /
      REQUIRED_FIELD
else compute quotient =
      (data_state_value (department_number_id) +
      REQUIRED_FIELD - 1) / REQUIRED_FIELD.

if !mod (quotient, 2) equal 1 then
   /* field is required */
else /* field is not required */ .
```

You can also use the subroutines `s$decode_flags` and `s$encode_flags` to read and set data-state values. For information on these routines, see the *VOS COBOL Subroutines Manual (R019)*.

If you want to alter the data states within the program, the `perform screen input` statement must both read from and write to the data-states array. If you include the

datastates form option within the perform screen input statement, that statement writes to the array for each form display. To make the perform screen input statement read from the array, you must turn on the COPY_DATASTATE switch for the form. To set this switch, you must use the options form option.

The options form option allows you to specify a set of 32 switches for the form. The VOS include file (master_disk)>system>include_library>form_options.incl.cobol defines mnemonic constants for these switches:

```
%replace SPECIAL_OPTION_30      by 1073741824 /* 2**30 */
%replace SPECIAL_OPTION_29      by 536870912  /* 2**29 */
%replace SPECIAL_OPTION_28      by 268354456   /* 2**28 */
%replace VALIDATE_ONE_FIELD     by 32768
%replace CHECK_3270_FORMS_MODEL by 8192
%replace COPY_DATASTATE        by 256
%replace NO_COPY_UPDATE        by 128
%replace VALIDATE_ERRORS_OFF    by 64
%replace WIDE_CURSOR           by 4
%replace VERTICAL_SCROLL_TRAP   by 2
```

The meaning of each switch is described in the discussion of the options form option in Chapter 5, "Form Options."

You can include the form_options.incl.cobol file in your program and define the set of form switches as follows:

```
copy 'form_options.incl.cobol'.
```

```
01 emp_info_options      comp-5.
```

Before reading emp_info_options in a screen statement, you must initialize it. You can specify values for some of the form switches within the Forms Editor. To initialize those switches to the values specified in the Forms Editor, execute the perform screen inquire statement after the perform screen initialization statement.

```
perform screen inquire with formid (employee_form_id)
options (emp_info_options) status (error_code).
```

Remember to check the returned status code.

You can then turn on the COPY_DATASTATE switch as follows:

```
add COPY_DATASTATE to emp_info_options.
```


You can then include the options and datastates form options within the perform screen input statement as follows:

```
perform screen input 'employee_info' update (employee_fields)
  with formid (employee_form_id) beep (beep_switch)
  keyused (key_code) options (emp_info_options)
  datastates (emp_data_states) putcursor (cursor_field)
  status (error_code).
```

If the COPY_DATASTATE switch is true, the perform screen input statement reads the data-state table and applies any changes to the fields before displaying the form. When the user submits the form, the perform screen initialization statement writes any changes in the data state to the table.

For example, to determine whether the user gave a value for the department_number field, check the value of the FIELD_VALUE_GIVEN switch within emp_data_state (department_number_id). If the switch is true, then a value was given; if the value is false, then the field value is null.

If you want to make the department_number field required, set the value of the REQUIRED_FIELD switch of emp_data_state (department_number_id) to true before displaying the form.

- * Check current value of required switch. If it is false,
- * set it to true.

```
if data_state_value (department_number_id) >= 0 then
  compute quotient =
    data_state_value (department_number_id) /
    REQUIRED_FIELD
else compute quotient =
  (data_state_value (department_number_id) +
  REQUIRED_FIELD - 1) / REQUIRED_FIELD.

if !mod (quotient, 2) not equal 1 then
  add REQUIRED_FIELD to
    emp_data_state (department_number_id).
```

You can change this value between form displays so that a field is required in some cases and not in others.

Changing a Display Type

In the sample application, the display type of each field is defined in the Forms Editor and never altered. You can, however, assign a new display type to a field at any time within the program.

A display type consists of several field visual attributes (such as intensity and justification), several field action attributes (such as traps and auto-tab), and other attributes such as a field picture, a cycle list, or a range of valid values. For an explanation of each display-type attribute, see Chapter 7, "Display Types."

The visual attributes of a display type are indicated by a set of 32 switches. The action attributes are indicated by another set of 32 switches. The VOS include file (master_disk)>system>include_library>form_displaytype.incl.cobol defines mnemonic constants for these switches.

Display types defined within the Forms Editor are called *predefined display types*. You cannot alter predefined display types, but you can alter the display-type characteristics of a field by defining a new display type and assigning it to the field. For example, in the sample application you can make the `employee_number` field blink by creating a blinking display type and assigning it to that field.

Every display type has a unique display-type ID. Predefined display types have negative IDs. You can define display types with IDs in the range of 11 to 16,383, inclusive.

You can define a display type in a display-type clause of the perform screen initialization, perform screen input, or perform screen output statement. Alternately, you can define the display type in a display-type description clause of a perform screen update statement. The following example defines a new display type for the `employee_number` field in the sample application.

```
%replace LOW_BLINKING_DT      by 11

copy 'form_displaytype.incl.cobol'.
.
.
.
perform screen update with status (error_code),
    giving displaytype (LOW_BLINKING_DT)
        visual (LOW_INTENSITY_VISUAL, BLINKING_VISUAL)
        picture ('zzzzz9').
```

The new display type retains the low-intensity attribute and the picture from the predefined display type of the `employee_number` field while adding the blinking attribute. Note that the perform screen update statement only defines the display type. It does not assign it to any field.

Within a screen statement, you can reference the display type of a specific field with the `displaytype` field option, or you can reference the display types for all fields in the form with the `displaytypes` form option. This section describes the use of the `displaytypes` form option. For more information on the `displaytype` field option, see Chapter 6, "Field Descriptions."

To reference the display types of a form, first declare a table of display-type IDs. The table must contain one element for each field in the form, and each element must be a two-byte integer. You can use the value *form_name_max_ids* from the form-IDs file to define the extent of the table. The following example declares a table of display-type IDs for the sample application.

```
01 emp_dt_ids.
   02 dt_id_value          comp-4 occurs employee_info_max_ids.
```

This example assumes that the *employee_info_ids.incl.cobol* include file is referenced earlier in the program.

You can initialize the *emp_dt_ids* table to the predefined display-type IDs by referencing it in the *displaytypes* form option in the *perform* screen initialization statement as follows:

```
perform screen initialization 'employee_info'
      into (initial_fields)
      with formid (employee_form_id) displaytypes (emp_dt_ids)
      status (error_code).
```

You can then change the display type of a field by changing the associated value in the table. For example, to assign the newly defined display type *LOW_BLINKING_DT* to the *department_number* field, execute the following assignment.

```
move LOW_BLINKING_DT to dt_id_value (department_number_id).
```

Caution: If you might want to subsequently change back to the predefined display type, you should save the original display-type ID before executing this assignment.

For the new display type to take effect in the form display, you must reference the *emp_dt_ids* table in a *displaytypes* form option within the *perform* screen input statement as follows:

```
perform screen input 'employee_info' update (employee_fields)
      with formid (employee_form_id) displaytypes (emp_dt_ids)
      beep (beep_switch) keyused (key_code)
      putcursor (cursor_field) status (error_code).
```

You can assign a different display type to any field prior to subsequent form displays. You can also alter any display types that you have defined. These changes are automatically applied to every field that has that display type. For more information on display types, see Chapter 7, "Display Types."

Forms Input Mode

If all input for an application (or a part of an application) is performed through a series of form displays, consider putting the input port into forms input mode. In this mode, all input typed by the user is assumed to be forms input.

If a port is in forms input mode and VOS receives input characters when a form is not displayed, the characters are saved until a form is displayed. The characters are then applied to that form. This allows the user to anticipate the next form and begin typing input before the form is displayed.

If the port is not in forms input mode, any characters typed between form displays are echoed to the screen (possibly corrupting a subsequent form display) and are not applied to the next form display.

For more information on forms input mode, see Chapter 14, “Subroutines,” and Appendix F, “Global Control Operations.” The `cobol` command is described in the *VOS Commands Reference Manual (R098)*.

Compiling and Binding the Application

You can compile an FMS program in the same way that you would compile any program. No special options are necessary. If you use the include file produced by the Forms Editor, you must ensure that the directory containing those files is in your include library search library paths. If you use include files from the system `include_library` directory, you must ensure that directory is in your search paths.

To compile the sample program, you can invoke the COBOL compiler as follows:

```
cobol enter_employees
```

The `cobol` command is described in the *VOS Commands Reference Manual (R098)*.

When you bind the application, you must ensure that the form object module created by the Forms Editor is in your object search paths. You must also include the following directory in your object library search paths:

```
(master_disk)>system>icss_fms_object_library
```

To bind the sample application, you can invoke the binder as follows:

```
bind enter_employees employee_info -search >system>icss_fms_object_library
```

The `bind` command is described in the *VOS Commands Reference Manual (R098)*.

The name of the application program module created by the binder is `enter_employees.pm`.

PART 2: FMS Reference Guide

)

)

)

)

)

Chapter 3:

The Elements of FMS

This chapter describes basic concepts and components of FMS screen forms. The first section introduces the components of an FMS form. The second section describes field characteristics. The third section describes how International Character Set Support (ICSS) is supported for FMS applications. The final section describes how forms interact with an application program.

Form Components

A form consists of three things.

- General form attributes
- Background text
- Fields

The general form attributes are set and modified with the Forms Editor `(MENU) S` request and with the form options within screen statements. For information on the Forms Editor, see Chapter 4, “The Forms Editor.” For information on the form options, see Chapter 5, “Form Options.”

The Forms Editor allows you to establish background text to guide the user through the fields of the form. For more information, see Chapter 4, “The Forms Editor,” and Chapter 2, “Developing an FMS Application.”

The fields are the most important element of most forms. The following section describes how fields are defined and discusses some of the characteristics of fields.

Fields

You can initially define fields in the Forms Editor or in an application program. Each field in a form is assigned a unique integer *field ID*.

Within the Forms Editor, you can define fields using the `(MENU) F` request. For more information, see Chapter 4, “The Forms Editor.”

Within a program, you can create, modify, and delete fields with the field description clause of the screen statements.

Chapter 6, “Field Descriptions,” explains the field characteristics you can specify or modify in field descriptions. Two of these characteristics are described further in separate chapters: display types in Chapter 7, “Display Types,” and data states in Chapter 8, “Data States.”

The remainder of this section describes some general characteristics of fields. The subsections discuss the following topics.

- Numeric fields versus alphanumeric fields
- Input fields versus output fields (and output-only fields)
- Null field values
- Initial output values

Numeric and Alphanumeric Fields

Fields can be categorized in many ways. Two important categories are numeric and alphanumeric fields. All fields can be classified as either numeric or alphanumeric.

A field is *numeric* if either of the following is true.

- The field has no associated picture, and the field data type is numeric.
- The field has an associated picture, and that picture contains only numeric picture characters.

Table 3-1 lists the numeric picture characters.

Table 3-1. The Numeric Picture Characters

Picture Character	Meaning
9	Allow a digit or hyphen (negative sign). Do not suppress leading zero.
Z or z	Allow a digit or hyphen (negative sign). Suppress a leading zero.
.	Fix the location of the decimal point. [†]
,	Group digits in large numbers. [†]

[†] The meanings of the period (.) and comma (,) picture characters are reversed if you set the `decimal is comma` option in the Forms Editor. The `decimal is comma` option is described in Chapter 4, “The Forms Editor.”

For more information on field pictures, see Chapter 9, “Field Pictures and Filtering.”

A field is *alphanumeric* if either of the following is true.

- The field has no associated picture, and has a non-numeric field-value variable.
- The field has an associated picture containing alphanumeric picture characters.

Table 3-2 lists the alphanumeric picture characters.

Table 3-2. The Alphanumeric Picture Characters

Picture Character	Meaning
-	Insert a literal hyphen.
/	Insert a literal slant.
B or b	Insert a literal space.
9 [†]	Allow a decimal digit. Do not suppress a leading zero.
A or a	Allow a letter or a space.
X or x	Allow a letter, a digit, or a space.
L or l	Allow a letter, a digit, or a space. Convert a letter to lowercase.
U or u	Allow a letter, a digit, or a space. Convert a letter to uppercase.

[†] The 9 picture character can appear in both numeric and alphanumeric pictures. A picture that contains only 9 characters is alphanumeric.

For more information on field pictures, see Chapter 9, “Field Pictures and Filtering.”

By default, numeric fields are right-justified, and alphanumeric fields are left-justified. You can override the default field justification with the `CENTER_FIELD_DATA`, `RIGHT_JUSTIFY_FIELD_DATA`, and `LEFT_JUSTIFY_FIELD_DATA` display-type visual switches. The visual switches are described in Chapter 7, “Display Types.”

Input Fields and Output Fields

At any point in time, each field in a form is either an input field or an output field. An *input field* is a field that the user can position to. Typically, the user can change the value of an input field. An *output field* is a field that the user cannot position to.

The `INPUT_FIELD` data-state switch must be true for an input field. Setting this switch to false changes the field to an output field.

In the Forms Editor you can define a field as input, output, or output only. If you specify input, the Forms Editor creates an input field. If you specify either output or output only, the Forms Editor creates an output field. If you specify output, an application program can change the field to an input field by setting the `INPUT_FIELD` data-state switch to true. If you specify output only, an application

cannot change the field to an input field. The value of the INPUT_FIELD data-state switch is ignored for output-only fields.

For information on the INPUT_FIELD data-state switch, see Chapter 8, “Data States.”
For information on the Forms Editor, see Chapter 4, “The Forms Editor.”

The default video display attributes for output fields differ from those of input fields. For output fields, the defaults are low intensity and no underline. For non-cycle input fields, the defaults are high intensity and underlined. For cycle input fields, the defaults are high intensity and no underline.

Null Field Values

A field's *null value* is the value displayed when the field is empty.

A field's null value is a string of spaces the length of the field if either of the following is true.

- The field does not have an associated picture.
- The field's picture does not include any of the following self-insertion characters: decimal point, slant, or hyphen.

If a field's picture contains self-insertion characters, those characters appear in the null value. If a numeric picture contains any 9 characters, zeros appear in those positions of the null value.

Table 3-3 lists some field pictures and the associated null field value. Note that digit-grouping characters never appear in the null value.

Table 3-3. Sample Field Pictures and Corresponding Null Values

Picture	Null Value
none	' '
XXLLUU	' '
999999	' '
X-X/XX	' - / '
ZZZ.99	' .00'
ZZ,ZZZ	' '

For information on pictures, see Chapter 9, “Field Pictures and Filtering.”

If a field is required, the form cannot be submitted if the field contains its null value. For information on required fields, see Chapter 8, “Data States.”

A cycle field can contain its null value only if the null value is in the cycle list. For information on cycle fields, see Chapter 7, "Display Types."

Initial Output Values

A field's *initial output value* is the value displayed in a field when the form is first displayed by the Forms Processor (as the result of executing an accept, perform screen input, or perform screen output statement).

The following rules determine the initial output value when screen statements are used. The rules for the accept statement are different and are given in Appendix A, "The accept Statement."

1. If a field description in the perform screen input or perform screen output statement specifies an initial field value, then that value is the initial output value.
2. If Rule 1 does not apply, and the perform screen input or perform screen output statement contains the update form option, and the NO_COPY_UPDATE options switch is false for the form, then the value of the field-value variable specified in the update option is the initial output value. You can initialize this value in the into form option of the perform screen initialization statement. The value stored by the into option is discussed later in this subsection. The value can subsequently be altered if referenced in the update option of a perform screen update statement that includes a field description specifying a different initial value.
3. If neither Rule 1 nor 2 applies, and a field description in a perform screen update statement between the perform screen initialization statement and the perform screen input or perform screen output statement specifies an initial field value, then that value is the initial output value.
4. If Rules 1 through 3 do not apply, and the form is alterable by accept, and a field description in the perform screen initialization statement specifies an initial field value, then that value is the initial output value.
5. If Rules 1 through 4 do not apply, and an initial field value is specified in the (MENU) F request within the Forms Editor, then that value is the initial output value.
6. If Rules 1 through 5 do not apply, and the field is a cycle field, then the first value in the cycle list is the initial output value.
7. If Rules 1 through 6 do not apply, then the null value for the field is the initial output value.

If the perform screen initialization statement includes the into form option, the field-value variables referenced in that option are initialized. In this case, the following rules determine the initial value of each field.

1. If the form is alterable by accept, and the perform screen initialization statement includes a field description that specifies an initial value for the field, then that value is used.
2. If Rule 1 does not apply, and an initial value for the field is specified in the (MENU) F request of the Forms Editor, then that value is used.
3. If neither Rule 1 nor 2 applies, and the field is a cycle field, then the first value in the cycle list is used.
4. If Rules 1 through 3 do not apply, then the field's null value is used.

For more information on the into and update form options, see Chapter 5, "Form Options." For information on the NO_COPY_UPDATE options switch, see the description of the options form option in Chapter 5, "Form Options."

For information on field descriptions and the initial field option, see Chapter 6, "Field Descriptions."

For information on the ALTERABLE BY ACCEPT option, see the discussion of the (MENU) s Forms Editor request in Chapter 4, "The Forms Editor." Also, for information on the (MENU) F Forms Editor request, see Chapter 4, "The Forms Editor."

International Character Set Support

This section describes how to provide International Character Set Support (ICSS) within an FMS application.

VOS supports the following character sets.

- ASCII
- Latin alphabet No. 1
- Kanji
- Katakana
- Hangul

The non-ASCII character sets are called *supplemental* character sets. The supplemental set that is currently in use is sometimes referred to as the *right graphic set* because it is located on the right half of the internal code page (composed of bytes in the range from 160 to 255 decimal).

Within a program, the character sets are typically referred to by integer IDs, as follows:

Character Set ID	Character Set Name
0	ASCII_CHARACTER_SET
1	LATIN_1_CHAR_SET
2	KANJI_CHAR_SET
3	KATAKANA_CHAR_SET
4	HANGUL_CHAR_SET

Constants for these values are defined in the file
(master_disk)>system>include_library>char_sets.incl.cobol.

Specifying the Character Set for a Field

At most, each field of an FMS form can support ASCII, katakana, and one other supplementary character set. Optionally, you can limit a field to any single character set by using the charset display-type option. The charset option takes one operand, a character-set ID.

You can set an initial value for the charset option for each predefined field in the Forms Editor.

For more information on the charset option, see Chapter 7, "Display Types."

Storing ICSS Strings

Within program variables, character-string values that contain characters from supplemental character sets can be stored in several different formats.

Each supplemental character might be preceded by a special control character called a *single-shift* character. A single-shift character indicates to which character set the following character belongs.

Sometimes, a default character set is established for a string. Characters from the default character set do not require a single-shift character. That is, any supplemental character in the string that is not preceded by a single-shift is assumed to be of the default character set. If a string contains many characters from a specific supplemental set, omitting the single-shift characters for that set can significantly reduce the storage size of the string.

When the Forms Processor transfers a character-string value from a field-value variable to a field, it must correctly interpret any supplemental characters in the string. Similarly, when the Forms Processor transfers a field value to a character-string field-value variable, it must put the correct single-shift characters in the stored

value. You can use the `shift` and `unshift` field options to indicate how you want supplemental characters translated.

The `shift` and `unshift` field options each take one operand: a character-set ID. The `shift` option indicates that all supplemental characters except those of the specified character set must be preceded by single-shifts in the field-value variable. The `unshift` option indicates that the field-value variable contains no shifts and that all supplemental characters in the string are assumed to be of the specified character set. The Forms Processor uses the information provided by the `shift` or `unshift` option to determine how to interpret the variable value on input to the form and how to store the value on output from the form.

For more information on the `shift` and `unshift` field options, see Chapter 6, “Field Descriptions.”

3270 Device Dependencies

Some FMS features behave differently under different configurations. In particular, because the 3270 is a block-mode terminal, FMS behavior under 3270 Support is somewhat unusual.

The following features are not fully supported on 3270 terminals.

- Traps
- Right-justified fields
- Required fields
- Cycle fields
- Field pictures
- Window fields
- `NEW_DATA_IN_FIELD` data-state switch
- Disappearing default
- Blinking, underlining, and inversion attributes

Traps are not handled at all by 3270 terminals. Right justification, field pictures, required fields, cycle fields, and window fields are partially supported. Test the behavior of these features before using them in a 3270 application.

If you set the `CHECK_3270_FORMS_MODEL` options switch for a form, an error is indicated if the form uses any critical features that are not supported by 3270 terminals. The use of convenience features such as blinking fields and disappearing defaults is not diagnosed as an error. For information on the options form option, see Chapter 5, “Form Options.”

For further information on 3270 Support, see *VOS Communications Software: 3270 Support and 3270 Emulation (R026)*.

Forms and the Application Program

This section describes how an application program interfaces with screen forms.

An application program can deal with just one form or with several forms simultaneously. A program can have several *active* forms, each identified by a form ID that is unique within the program. An active form is stored in the user heap and can be modified or displayed by the program.

More information on using multiple active forms appears under the headings “Initializing a Form,” “Saving a Form,” and “Discarding a Form” in the following subsection.

The first of the following subsections describes the operations an application program can execute on a form. The second subsection describes how and when control transfers between the application and the Forms Processor.

Operations on Forms

This subsection describes the operations you can execute on a form:

- defining the form
- initializing the form
- displaying the form (and optionally accepting user input)
- modifying the form
- getting information about the form
- saving the form
- discarding the form.

Defining a Form. You can define a form either in the Forms Editor or in a perform screen initialization statement. If possible, you should define all forms in the Forms Editor. Defining a form within the application program is more difficult and less efficient. For information on the Forms Editor, see Chapter 4, “The Forms Editor.”

Within an application, you invoke a form predefined in the Forms Editor by referencing it in the form specifier of a perform screen initialization statement. If you do not invoke a predefined form, then the form options, field descriptions, and display-type descriptions in the perform screen initialization statement define the form. For more information on defining a form within the perform screen initialization statement, see Chapter 6, “Field Descriptions.”

You can also define a form in the obsolete accept statement. For more information, see Appendix A, “The accept Statement.”

Initializing a Form. You initialize a form with the perform screen initialization statement. When you initialize a form, an internal representation of the form, called the *display list*, is created.

The perform screen initialization statement can reference a predefined form, or it can define the form. In the latter case, the perform screen initialization statement both defines and initializes the form.

The perform screen initialization statement does **not** display the form.

When a form is initialized, it becomes active. It also becomes the current form for the window in which it is initialized.

For more information on the perform screen initialization statement, see Chapter 16, "Statements."

You can also initialize a form with the obsolete accept statement. For more information, see Appendix A, "The accept Statement."

Displaying a Form. You display a form with either the perform screen input or perform screen output statement. The perform screen input statement invokes the Forms Processor to display the form and wait for the user to submit or cancel the form (or for some other action to cause return to the application). The perform screen output statement invokes the Forms Processor to display the form, but does not accept input from the user.

Information on submitting and canceling forms and on the flow of control between a form and an application appears later in this chapter.

A form must be initialized before it can be displayed.

You can specify which form you want to display with the `formid` form option in the perform screen input or perform screen output statement. If you omit this option, the most recently referenced form is displayed.

For more information on the perform screen input and perform screen output statements, see Chapter 16, "Statements."

The obsolete accept statement can also display a form. For more information, see Appendix A, "The accept Statement."

Modifying a Form. You can modify a form with form options, field descriptions, and display-type descriptions in the perform screen initialization, perform screen input, or perform screen output statement. In addition, two special screen statements are provided exclusively for modifying an active form: perform screen update and perform screen delete.

The perform screen update statement allows you to specify the following:

- form options to change the form's appearance and behavior
- field descriptions to add new fields or change the attributes of existing fields
- display-type descriptions to define new display types or modify existing display types.

The form options are described in Chapter 5, "Form Options." Field descriptions are described in Chapter 6, "Field Descriptions." Display-type descriptions are described in Chapter 7, "Display Types."

The perform screen update statement changes the internal image of the form (the display list), but it does **not** display the form. Changes you specify in a perform screen update statement become visible to the user the next time a perform screen input or perform screen output statement is executed.

The perform screen delete statement allows you to delete fields, display types, or both from a display list. You can delete a display type only if no field in any active form references it. If you delete a field, that field is absent when the form is next displayed.

For more information on the perform screen update and perform screen delete statements, see Chapter 16, "Statements."

Getting Information about a Form. The perform screen inquire statement returns information about an active form. The perform screen inquire statement has the following parts:

- form options to return general information about the form
- field descriptions to return information about specific fields
- display-type descriptions to return information about specific display types.

Many form options, field options, and display-type options that provide information to the form in other statements return information **about** the form in the perform screen inquire statement. In addition, some options are provided for use only in the perform screen inquire statement.

For more information on form options, see Chapter 5, "Form Options." For information on field descriptions, see Chapter 6, "Field Descriptions." For information on display-type options, see Chapter 7, "Display Types."

For more information on the `perform screen inquire` statement, see Chapter 16, “Statements.”

Saving a Form. You can use the `perform screen save` statement to prevent a form from becoming inactive.

Normally, if a form has been displayed, and another form is then displayed in place of it in the same window, the first form becomes inactive. This means that to redisplay the first form, you must re-initialize it with the `perform screen initialization` statement. Because form initialization consumes time and resources, it is usually better to save the form if you expect to display it again.

Saving a form with the `perform screen save` statement is referred to as *caching* the form.

The `perform screen save` statement increments an internal form reference count. The form remains active as long as its reference count remains positive.

For further information on form reference counts and form caching, see Chapter 12, “Form Caching.”

Discarding a Form. You can discard a cached form by using the `perform screen discard` statement. Discarding a form frees heap space.

The `perform screen discard` statement decrements the internal form reference count. If the reference count becomes 0 or negative, the form is deactivated and its heap space is freed.

Note that one `perform screen discard` statement negates exactly one `perform screen save` statement.

For further information on form reference counts and form caching, see Chapter 12, “Form Caching.”

Control Transfer between an Application and a Form

When an application program executes a `perform screen input` statement, the Forms Processor is invoked to display the specified form. The Forms Processor maintains control while the user manipulates the form. Control returns from the Forms Processor to the program when any of the following occur.

- The user submits the form.
- The user cancels the form.
- A trap occurs.
- The time-out period for the form expires.
- The form is knocked down by another process or task.
- The form is output-only.
- Certain errors or other unusual situations occur.

Each of these cases is discussed later in this subsection. In each case, when control returns from the Forms Processor, program execution continues with the statement following the `perform screen input` statement.

Form Submission. A user can submit a form by pressing the `(ENTER)` key or by pressing any function key designated as an entry key in the `maskkeys` form option.

When a form is submitted, the Forms Processor validates that each field value conforms to the field display type. First, if the field's display type has an associated picture, the Forms Processor checks that the field value conforms to that picture. Next, if the display type has a restriction on the range of valid values, the Forms Processor checks that the field value is in that range. Finally, if the display type has an associated validation routine, that routine is invoked to test the field value.

If no errors occur, the values in the form are converted and stored into the program variables specified in the `update form` option or `update field` options in the `perform screen input` statement. Control then returns to the application, and execution continues with the statement following the `perform screen input` statement.

If a validation error does occur, control does **not** return to the application. Instead, the Forms Processor redisplay the form, sounds the terminal bell, positions the cursor to the field that failed validation, and displays an error message to the user. The user can then either correct the error and submit the form again, or cancel the form. Each time the user submits the form, the full validation suite for each field is performed.

For further information on field validation and validation routines, see Chapter 10, "Error Handling and Field Validation." See also the descriptions of the picture and range display-type options in Chapter 7, "Display Types."

Form Cancellation. The user can cancel the form by pressing the CANCEL key or by pressing any function key designated as a cancel key in the `maskkeys` form option.

When a form is canceled, the field values in the form are not returned to the application. No field validation is performed, and field values are **not** loaded into program variables.

If the `perform` screen input statement includes the `keyused` form option, the value `-1` is returned in that option when the form is canceled. For more information on the `keyused` option, see Chapter 5, "Form Options."

If the `perform` screen input statement does not contain the `keyused` option, but does contain the `status` form option, the error code `e$form_aborted (1453)` is returned in the `status` option when the user cancels the form. The statement following the `perform` screen input statement should normally check the value returned by the `status` option and take appropriate action. For more information on the `status` option, see Chapter 5, "Form Options."

If the `perform` screen input statement contains neither the `keyused` nor the `status` option, the error condition is signaled when the user cancels the form. For information on signaled conditions, see the description of the subroutine `s$enable_condition` in *VOS COBOL Subroutines Manual (R019)*.

Traps. A *trap* is a return from the form to the application caused by the user moving the cursor. Two kinds of traps can be defined on individual fields.

- Trap on field entry
- Trap on field exit

In addition, you can define a vertical scroll trap on a form.

You can establish the individual field traps by setting the `TRAP_ON_FIELD_ENTRY` and `TRAP_ON_FIELD_EXIT` display-type action switches. You can establish a vertical scroll trap by setting the `VERTICAL_SCROLL_TRAP` switch in the options form option.

If the `TRAP_ON_FIELD_ENTRY` action switch is true for a field, then when the user positions the cursor to that field, control returns to the program. If the `perform` screen input statement includes the `keyused` form option, the value `-9` is returned in that option when a trap on field entry occurs.

If the `TRAP_ON_FIELD_EXIT` action switch is true for a field, then when the user moves the cursor out of that field, control returns to the program. If the `perform` screen input statement includes the `keyused` form option, the value `-2` is returned in that option when a trap on field exit occurs.

If the `VERTICAL_SCROLL_TRAP` options switch is true, then control returns to the application whenever the user tries to move the cursor beyond the last field in the form or before the first field in the form. (If the `VERTICAL_SCROLL_TRAP` switch is

false, and the user attempts to move beyond the last field in the form, the cursor moves to a field at the top of the form; if the user attempts to move before the first field in the form, the cursor moves to a field at the bottom of the form.)

When a trap occurs, field values from the form are validated and loaded into the program variables. You can limit the fields that are validated by setting the `VALIDATE_ERRORS_OFF` or `VALIDATE_ONE_FIELD` switch in the options form option.

When a trap occurs, control always transfers to the statement following the `perform screen input` statement. Typically, the application updates the form (based on the cursor position or field values already given) and redisplay the form.

The `nextcursor` form option returns the logical position for the cursor in the redisplayed form.

For a trap on field exit or on field entry, the `getcursor` form option returns the field ID of the trap field. For a vertical scroll trap, the `getcursor` option returns the field ID of the field from which the cursor is being moved.

Form Time-Out Period. The form *time-out period* is the maximum amount of time the Forms Processor waits for the user to submit or cancel a form. If the user does not submit or cancel the form before the time-out period expires, the form is automatically canceled, and the error code `e$timeout` (1081) is returned in the `status` form option. Form cancellation is discussed earlier in this section.

The `timeout` form option specifies the time-out period for a form. For a predefined form, you can specify an initial value for this option in the `(MENU) s` Forms Editor request. If you do not specify a time-out period, or if you specify a period of -1, the Forms Processor waits indefinitely for the user to either submit or cancel the form.

For further information on the `timeout` and `status` options, see Chapter 5, "Form Options." For information on the `(MENU) s` Forms Editor request, see Chapter 4, "The Forms Editor."

Form Knocked Down. A form can be *knocked down* by another process or another task that needs to use the same output device. A process or task knocks down a form by calling the subroutine `s$control` with opcode 240, `KNOCK_DOWN_FORM_OPCODE`, or opcode 264, `KNOCK_DOWN_FORM_OK_OPCODE`.

For further information on these operations, see Appendix F, “Global Control Operations.”

If a form is knocked down, field values are returned without any kind of validation. The value -6 is returned in the `keyused` form option. If you want the application to behave appropriately when the form is knocked down, you should specifically check for this value.

For information on the `keyused` option, see Chapter 5, “Form Options.”

Note: Knocking down a form requires cooperation between the process or task that displays the form and the process or task that knocks it down. This operation is most common in tasking applications.

Output-Only Forms. An *output-only* form is a form with no input fields and no keys masked. If you attempt to display such a form with the `perform screen input` statement, the form is not displayed; instead, control returns immediately to the application. If the `perform screen input` statement includes the `keyused` option, the value -1 is returned in that option. If the statement includes the `status` option, the error code `e$form_needs_input_field` (3918) is returned in that option.

You should use the `perform screen output` statement to display output-only forms.

Other Situations. Other situations that can cause premature return from a `perform screen input` or `perform screen output` statement include device, channel, and network problems. For more information on these cases, see Chapter 10, “Error Handling and Field Validation.”

Chapter 4:

The Forms Editor

This chapter describes the Forms Editor used to create, modify, and save FMS forms. The following topics are discussed:

- overview of the Forms Editor
- the `icss_edit_form` command that invokes the Forms Editor
- definitions of Forms Editor terms
- entering background text
- the Forms Editor edit requests
- the files produced by the Forms Editor.

Overview of the Forms Editor

The Forms Editor is an interactive screen editor, similar to the Word Processing Editor. The text you enter and modify is displayed on your terminal screen so that you always see an up-to-date version of the text. When you change the text, the Forms Editor immediately updates the screen, showing the revised version.

The Forms Editor lets you build a full-scale model of a form on your terminal screen. The Forms Editor enables you to do the following:

- define input and output fields, choosing from a variety of action and display options for each field
- type background text to provide instructions and field titles
- move fields in the form and alter their characteristics
- view and test the latest version of a form during the editing session.

A form constructed with the Forms Editor is called a *predefined* form because it is defined before the application is run. Because some form options and field definitions in the screen statements can override those in the predefined form, you can modify the form in an application.

The Form Design Process

The design process can be generalized as follows:

- Set the form options using the **(MENU) S** (Set/modify form options) Forms Editor request.
- Type in background text.
- Insert form fields using the **(MENU) F** (Add/modify field) Forms Editor request.
- Test the form using the **(MENU) X** (Show exact form) Forms Editor request.
- Write the form definition, object module, and include files using the **(MENU) W** (Write) Forms Editor request.

The Two Forms Editors

VOS supports two versions of the Forms Editor. The command `edit_form` invokes the older version. This version is supported for compatibility with existing forms applications. The command `icss_edit_form` invokes the newer version. This version supports many new features, including International Character Set Support (ICSS). You should use this version for the development of new forms applications.

The command interface to the two Forms Editors is almost identical. The `icss_edit_form` command has one additional argument: `-fortran_strings`. Other visible differences occur in the forms displayed by the **(MENU) F** and **(MENU) S** menu edit requests. The new Forms Editor also supports a new menu option: Insert literal (**(MENU) L**). The forms shown in this chapter are for the `icss_edit_form` command. The forms for the old Forms Editor are shown in Appendix B, “The `edit_form` Command.”

The form object modules produced by the two Forms Editors are **not** compatible.

The form definition file produced by the new Forms Editor contains options that are unknown to the old Forms Editor. Therefore, a form that has been created or modified with the new Forms Editor **cannot** be modified by the old Forms Editor. A form created with the old Forms Editor **can** be modified by the new Forms Editor, after which it can no longer be modified by the old Forms Editor.

You can create a new-style form object module from an old-style form definition file without modifying the form definition file by issuing the following command:

```
icss_edit_form old_form_name -no_edit
```

Because this form of the `icss_edit_form` command does not modify the form definition file, you can still edit that file with the old Forms Editor.

The `icss_edit_form` command is described in the next section. For more information on the `edit_form` command and the old Forms Editor, see Appendix B, "The `edit_form` Command."

The icss_edit_form Command

Purpose

The command `icss_edit_form` invokes the Forms Editor.

CRT Form

```
----- icss_edit_form -----
input_path: 
form_path: 
-into:          no
-prefix:        no
-library:       accept_field_definitions
-edit:          yes
-backup:        yes
-force_write:   no
-basic:         no
-cobol:         no
-fortran:       no
-fortran_strings: yes
-pascal:       no
-pl1:          no
-c:            no
-produce_symtab: yes
```

Lineal Form

```
icss_edit_form input_path
    [ form_path ]
    [-into ]
    [-prefix ]
    [-library field_definitions_directory_name ]
    [-no_edit ]
    [-no_backup ]
    [-force_write ]
    [-basic ]
    [-cobol ]
    [-fortran ]
    [-no_fortran_strings ]
    [-pascal ]
    [-pl1 ]
    [-c ]
    [-no_produce_symtab ]
```

Arguments► *input_path***Required**

The path name of an input form definition file. If the file exists, its name must have the suffix `.form`. You can omit the suffix when specifying the name in the command. If the file does not exist, the Forms Editor behaves as if the file is empty.

► `form_path`

An option specifying the file to which the edited form definition is to be written. If `form_path` does not have the suffix `.form`, the command adds that suffix. If you do not specify a value for `form_path`, it defaults to a file in the current directory with the same name as the file specified in the `input_path` option. If the specified file does not exist when you write out the form, the Forms Editor creates it.

The form being edited is given the simple name of the output form definition file without the suffix `.form`. A form name should not exceed 15 characters; otherwise, the names of some automatically generated include files may exceed 32 characters and be truncated.

Note: Do not give a form the same name as the program that displays it — both a form and its related program require uniquely named object modules.

► `-into`

(CYCLE)

An option to create a field-values file for each programming language specified by the language options. The Forms Editor names the field-values file (an include file) `form_name.incl.language` and puts the file in the current directory. You can override this argument with the Forms Editor (MENU) s request, described later in this chapter.

► `-prefix`

(CYCLE)

An option to add a prefix to each field-identifier name in any field-IDs file that the Forms Editor generates. The default prefix is the name of the form followed by an underline. You can specify a different prefix with the (MENU) s request in the Forms Editor. You can override this argument with the Forms Editor (MENU) s request, described later in this chapter.

If you choose the `-prefix` option, the Forms Editor also adds the prefix to each variable name in any VOS BASIC or VOS FORTRAN field-values file that it generates. The field-values files for other languages are not affected.

► `-library field_definitions_directory_name`

An option to specify a directory for storing and retrieving field definition files. The Forms Editor searches the directory for field definition files when you use the (MENU) R request and writes field definition files to the directory when you use the (MENU) E request. If you do not specify this option, the default value is a subdirectory of your current directory named `accept_field_definitions`. If the directory you specify, either directly or by default, does not exist when you issue a (MENU) E request, the Forms Editor creates the directory.

► **-no_edit** (CYCLE)

An option to create new language include files and a new object module from an existing form definition file without editing the form. If you specify the `-force_write` option with this option, the Forms Editor also writes a new form definition file. By choosing the `-no_edit` option, you can run the Forms Editor in either a batch process or a started process. If you do not use the `-no_edit` argument, the Forms Editor reads the form definition file, displays a representation of the form, and lets you edit it.

► **-no_backup** (CYCLE)

An option to specify that no backup file is created for the *input_path* file. If you do not use the `-no_backup` argument, and the *input_path* and *form_path* files are in the same directory, the Forms Editor renames the old file and gives it the name of the *input_path* file (including its suffix `.form`), with the suffix `.backup` added. The backup file is created each time you write out the form with the (MENU) W request; it replaces a previous backup file of the same name if one exists.

► **-force_write** (CYCLE)

An option to write a new form definition file (*form_name.form*) when you invoke `icss_edit_form` with the `-no_edit` option. If you do not use the `-force_write` argument, `-no_edit` produces the object module and specified include files only. Use `-force_write` with `-no_edit` to generate a `.backup` form file or to rename your form without re-editing it.

Note: Do not use the command `rename` to rename a form; object and include files must be renamed, and prefixes in include files need to be reassigned.

► **-basic** (CYCLE)

An option to create VOS BASIC versions of the field-IDs file and the field-values file. If you do not use the `-basic` argument, the Forms Editor does not create VOS BASIC versions of the files. You can override this argument with the Forms Editor (MENU) S request, described later in this chapter.

► **-cobol** (CYCLE)

An option to create VOS COBOL versions of the field-IDs file and the field-values file. If you do not use the `-cobol` argument, the Forms Editor does not create VOS COBOL versions of the files. You can override this argument with the Forms Editor (MENU) S request, described later in this chapter.

► `-fortran`

(CYCLE)

An option to create VOS FORTRAN versions of the field-IDs file and the field-values file. If you do not use the `-fortran` argument, the Forms Editor does not create VOS FORTRAN versions of the files. You can override this argument with the Forms Editor (MENU) s request, described later in this chapter.

► `-no_fortran_strings`

(CYCLE)

An option to not use the string data type in FORTRAN include files. The string type is a VOS FORTRAN extension that might not be compatible with other FORTRAN compilers. You can override this argument with the Forms Editor (MENU) s request, described later in this chapter.

► `-pascal`

(CYCLE)

An option to create VOS Pascal versions of the field-IDs file and the field-values file. If you do not use the `-pascal` argument, the Forms Editor does not create VOS Pascal versions of the files. You can override this argument with the Forms Editor (MENU) s request, described later in this chapter.

► `-pl1`

(CYCLE)

An option to create VOS PL/I versions of the field-IDs file and the field-values file. If you do not use the `-pl1` argument, the Forms Editor does not create VOS PL/I versions of the files. You can override this argument with the Forms Editor (MENU) s request, described later in this chapter.

► `-c`

(CYCLE)

An option to create VOS C versions of the field-IDs file and the field-values file. If you do not use the `-c` argument, the Forms Editor does not create VOS C versions of the files. You can override this argument with the Forms Editor (MENU) s request, described later in this chapter.

► `-no_produce_symtab`

(CYCLE)

An option to produce a form object module without a runtime symbol table. Because the forms runtime symbol table is small, use the `-no_produce_symtab` option only if there is a shortage of virtual memory.

Explanation

The `icss_edit_form` command invokes the Forms Editor. After you issue the `icss_edit_form` command, your process is at *edit request level*. At edit request level, you can enter text, or you can make a number of edit requests. These requests are described later in this chapter under the heading “Edit Requests.”

If you give the path name of an existing input form definition file when you issue the `icss_edit_form` command, the Forms Editor reads the file and displays a representation of the defined form.

The Forms Editor trims trailing spaces from all values you enter into the editor's request forms and from all lines you enter into the form you are constructing. It also deletes all empty lines from the bottom of the form. When you write the form definition file and the other files described previously, the files reflect these deletions.

If you choose the `-into` option when you invoke the Forms Editor, but do not specify any of the languages at that time, you can specify languages using the Forms Editor requests. If you do not specify any of the language options in the command line or in the Forms Editor, the `-into` option is ignored.

If you choose the `-into` option, the `-prefix` option or any of the language options (`-basic`, `-cobol`, `-fortran`, `-no_fortran_strings`, `-pascal`, `-pl1`, `-c`) for a particular form, these options are saved in the form definition file, and you do not have to respecify them with the Forms Editor requests or in future invocations of the Forms Editor on that form.

Access Requirements

You need read access to a form definition file to read it; you need write access to a form definition file, include file, or object module to write it.

Definitions

This section defines some terms used in describing the forms edit requests.

► **buffer**

A temporary storage area where the form you are editing is stored. This is sometimes referred to as the work area.




► **current line**

The line that currently contains the cursor. See Figure 4-1.

► **current word**

The word to which the cursor is currently positioned. See Table 4-1 and Figure 4-1.

Table 4-1. Explanation of Current Word

Cursor Position	Current Word	Description
 active	active	The cursor is on the first character of a word. The entire word is the current word.
act  ive	act	The cursor is in a word, but not at the first character. The characters to the left of the cursor comprise the current word.
active 	active	The cursor is not in a word. The word to the left of the cursor is the current word.

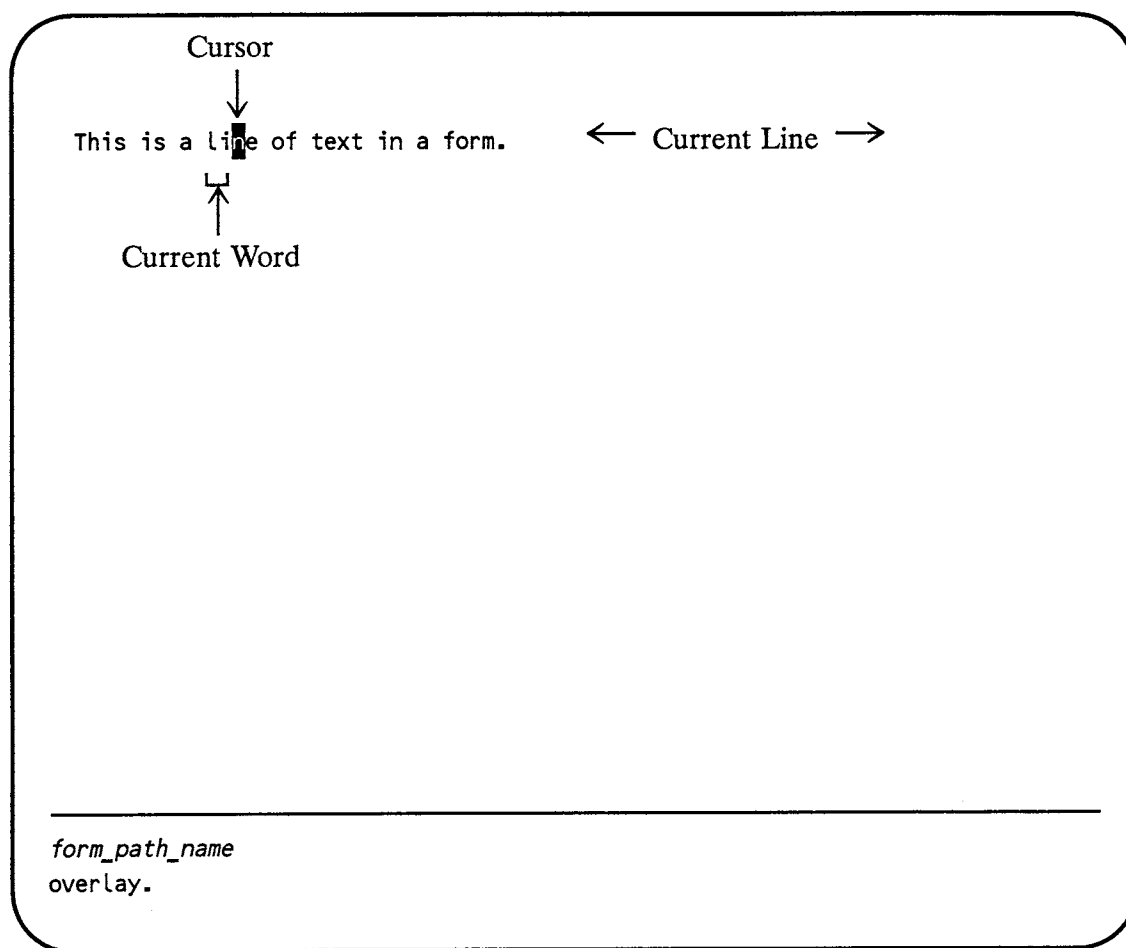


Figure 4-1. Forms Editor Terminology

► **screen**

The full video display area of your terminal.

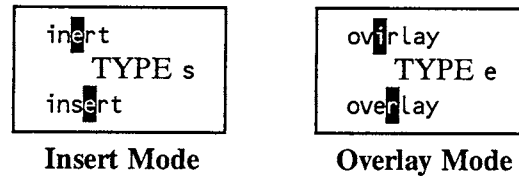
Note: Your form should not exceed the size of the screen on which it will be displayed.

Entering Text

Letters, digits, or other visible ASCII characters typed at edit request level are entered at the current position. This allows you to put background text in a form. Background text provides instructions, help, and field labels for the user. Like fields, background text **cannot** occupy the first character position of any line in the form. The first character position of each line is reserved for attribute information.

To enter characters from supplemental character sets in the background text, use the insert literal (MENU L) request described later in this chapter.

You can enter characters in two modes: insert and overlay. In *insert mode*, the Forms Editor enters the character you type at the current position and moves all subsequent characters on the current line to the right. In *overlay mode*, the Forms Editor replaces the character at the current position with the new typed character. In both cases, after entering a character, the cursor and the current position move one character position to the right. Figure 4-2 shows how insert and overlay modes work.



PD0002

Figure 4-2. Insert Mode and Overlay Mode

You can switch from one mode to the other with the En/disable overlay mode edit request (**MENU** 0) or with the **INSERT/OVERLAY** key, if one is defined for your terminal type.

The following keys also enter characters or change the way characters are entered.

- The **SPACE BAR** inserts or overlays a space at the current position.
- The **SHIFT** key simultaneously with another key puts in the uppercase version of the letter or the upper character shown on the key.
- The **ALPHA LOCK** key in its down (or locked) position causes all typed letters to appear in uppercase. In its up (or unlocked) position it has no effect. It has no effect when typing the non-letter keys.

Edit Requests

Edit requests can modify the text or the text display. For example, there are edit requests for changing the current position and for moving or deleting characters, words, or blocks of text.

You can issue some requests by using function keys on the keyboard. To issue other requests, you must select an item from a menu of requests. The **MENU** key displays the request menu. Unless you specify otherwise (with the **MENU** A request), the Forms Editor displays the menu each time you press the **MENU** key. In all cases, you make a menu request by typing a letter after pressing the **MENU** key. The uppercase and lowercase versions of a letter are equivalent when making menu requests.

The edit requests that you make directly are explained in the next subsection. The requests that you choose from the request menu are described in the subsequent subsection.

Press the **HELP** key for an online explanation of any Forms Editor request.

Direct Edit Requests

This subsection lists the direct edit requests available within the Forms Editor. The specific key mappings for each request depend on your terminal type. If you are not sure of the key mappings for your terminal, check with your system administrator. For information on defining a terminal type, see *VOS Communications Software: Defining a Terminal Type (R096)*.

- ▶ **←**
Moves the cursor left one column in the current line.
- ▶ **↓**
Moves the cursor down one line. The current column does not change.
- ▶ **↑**
Moves the cursor up one line. The current column does not change.
- ▶ **→**
Moves the cursor right one column in the current line.
- ▶ **BACK SPACE**
Moves the cursor back one position and deletes the character at the new cursor position. In insert mode, characters on the current line to the right of the new cursor position move left one column. In overlay mode, the Forms Editor puts a space at the new cursor position.
- ▶ **BACK TAB**
Moves the cursor to the right to the next column position established by text on a preceding line. This action is sometimes referred to as a *tab relative*.
- ▶ **BLANKS ←**
Moves the cursor left to the position immediately following the nearest visible character on the left. If no characters appear to the left of the cursor on the current line, this request moves the cursor to the first column of the line.

► **BLANKS** (→)

Moves the cursor to the nearest visible character right of the cursor in the current line. If no characters appear to the right of the cursor on the current line, this request moves the cursor to the end of the line.

► **CANCEL**

Cancels a multi-key request if you have not completed the sequence; cancels highlighting if you have enabled highlighting with a **MARK** request; or cancels a request that prompts you for input.

► **CHANGE CASE** (l)

Changes the letter at the cursor to lowercase. The cursor moves right one column.

► **CHANGE CASE** (↑)

Changes the letter at the cursor to uppercase. The cursor moves right one column.

► **COLUMN**

Enables column highlighting, if you have enabled highlighting with a **MARK** request. The mark and the cursor define the corners of a box, called a *column*. Pressing the **COLUMN** key highlights the column.

► **CYCLE**

Displays the next choice from a predefined set of values in a cycle field. Press the **CYCLE** key again to change the value. You can use this request within a cycle field in the forms displayed by the **MENU** F, **MENU** S, **MENU** V, and **MENU** X requests. These requests are described later in this chapter.

► **CYCLE BACK**

Displays the preceding choice from a predefined set of values in a cycle field. You can use this request within a cycle field in the forms displayed by the **MENU** F, **MENU** S, **MENU** V, and **MENU** X requests. These requests are described later in this chapter.

► **DEL**

Deletes the character at the cursor. In insert mode, characters to the right of the cursor move left one column, but the cursor does not move. In overlay mode, the Forms Editor puts a space at the cursor and the cursor moves right one column.

► **DELETE**

Deletes any highlighted text and pushes it onto the saved-text stack. In insert mode, characters right of the marked region move back to replace the deleted characters, and the cursor is set to the beginning of the moved text. In overlay mode, the Forms Editor replaces the deleted characters with spaces, and the cursor does not move.

► **DELETE** (←)

Deletes (insert mode) or replaces with spaces (overlay mode) all the characters in the current line to the left of the cursor. In insert mode, any text to the right of the cursor shifts left, and the cursor moves to the first column. In overlay mode, the cursor and any text to the right of the cursor do not move.

► **DELETE** (→)

Deletes all the characters in the current line to the right of the cursor. The cursor does not move.

► **DELETE** **(BLANKS)**

Deletes any space characters to the left, to the right, and at the cursor position in the current line. The cursor moves left to the nearest visible character, or to the beginning of the line, and the remaining characters move left to the new cursor position. This has the effect of joining a word after the cursor with a word before the cursor. **DELETE** **(BLANKS)** works the same way in both insert and overlay modes.

► **DELETE** **(RETURN)**

Deletes any blank lines above or below the current position.

► **DELETE** **(WORD)**

Deletes (insert mode) or replaces with spaces (overlay mode) the current word.

► **DISCARD**

Discards the block of text on the top of the saved-text stack. This operation is sometimes referred to as “popping the stack.”

► **(ENTER)**

Submits a form you have completed for any of the following menu edit requests:

Add/modify field	(MENU) F
Insert window field	(MENU) I
Set/modify form options	(MENU) S
Update fields	(MENU) U
Define/modify video display modes	(MENU) V
Show exact form	(MENU) X

When the form is submitted, the current buffer is redisplayed and the Forms Editor returns to edit request level.

► **(GO TO) (←)**

Moves the cursor to the first column in the current line.

► **(GO TO) (↓)**

Moves the cursor forward to the first column of the last line in the current region.

► **(GO TO) (↑)**

Moves the cursor backward to the first column in the first line of the current region.

► **(GO TO) (→)**

Moves the cursor to the end of the current line.

► **(GO TO) (COLUMN)**

Prompts you for a column number, and moves the cursor to that column in the current line.

► **(GO TO) (LINE)**

Prompts you for a line number, and moves the cursor to the first column in that line.

► **(GO TO) (LINE) (↓)** (or **(GO TO) (LINE) (RETURN)**)

Moves the cursor to the first column of the last line in the text buffer.

► **(GO TO) (LINE) (↑)**

Moves the cursor to the first column of the first line in the text buffer.

- ▶ **GO TO LINE RETURN** (or **GO TO LINE J**)
Moves the cursor to the first column of the last line in the text buffer.
- ▶ **GO TO MARK**
Exchanges the cursor and the mark, highlighting the text between.
- ▶ **HELP**
Lists the Forms Editor requests, then displays explanatory text for the Forms Editor request you specify.
- ▶ **INSERT DEFAULT**
Displays the default response to the current prompt. You can issue this request only when prompted for a response by a preceding request.
- ▶ **INSERT SAVED**
Inserts or overlays, at the cursor position, the block of text on the top of the saved-text stack.
- ▶ **INSERT SAVED DISCARD**
Inserts or overlays, at the cursor position, the block of text on the top of the saved-text stack, and then deletes it from the text and discards it from the saved-text stack. This allows you to examine and discard text from the top of the stack until you find the text you want. (This request is equivalent to the following sequence: **INSERT SAVED**, **GO TO MARK**, **DELETE**, **DISCARD**, **DISCARD**.)
- ▶ **LINE FEED**
Inserts a new line at the cursor without moving the cursor.
- ▶ **MARK**
Sets the mark at the cursor position. To highlight a region, set the mark at one end of the region, and move the cursor to the other end of the region.
- ▶ **MENU**
Displays the request menu, enabling you to make a menu request.
- ▶ **REDISPLAY**
Displays the current region again, thus removing any discrepancies between the actual text in an edit buffer and the displayed text.

► **REDISPLAY** **REDISPLAY**

This request sequence is the first remedy you should try for unusual terminal behavior. You can also use it to return the edit buffer to its current state after the terminal's power supply has been disconnected.

► **RETURN**

Inserts a new line at the current position. If you give the **RETURN** request from within the response to a prompt, the request issues the response.

► **SAVE**

Places the highlighted text onto the saved-text stack. This operation is also referred to as pushing text onto the stack.

► **SCROLL** **←**

Scrolls the displayed text left one screen column. The cursor moves left one screen column, maintaining its location in the text.

► **SCROLL** **↓**

Scrolls the displayed text down one line. The cursor moves down one screen line, maintaining its location in the text.

► **SCROLL** **↑**

Scrolls the displayed text up one screen line. The cursor moves up one screen line, maintaining its location in the text.

► **SCROLL** **→**

Scrolls the displayed text right one screen column. The cursor moves right one screen column, maintaining its location in the text.

► **SCROLL** **SHIFT** **↓**

Scrolls the displayed text down five screen lines. The cursor moves down five screen lines, maintaining its location in the text.

► **SCROLL** **SHIFT** **←**

Scrolls the displayed text left 20 screen columns. The cursor moves left 20 screen columns, maintaining its location in the text.

▶ **SCROLL** **SHIFT** **↑**

Scrolls the displayed text up five screen lines. The cursor moves up five screen lines, maintaining its location in the text.

▶ **SCROLL** **SHIFT** **→**

Scrolls the displayed text right 20 screen columns. The cursor moves right 20 screen columns, maintaining its location in the text.

▶ **SHIFT** **←** (or **WORD** **←**)

Moves the cursor left to the first letter of the current word or, when the cursor is at the beginning of the current word, to the first letter of the previous word. If you issue this request when the cursor is at or to the left of the beginning of the first word on the line, the cursor moves to the first column of the current line.

▶ **SHIFT** **↓**

Changes the cursor position to the region below the current region. The last line of the current region is included in the new region, redisplayed as the top line. After this request is executed, the new region becomes the current region.

▶ **SHIFT** **↑**

Changes the cursor position to the region above the current region. The first line of the current region is included in the new region, redisplayed as the last line. After this request is executed, the new region becomes the current region.

▶ **SHIFT** **→** (or **WORD** **→**)

Moves the cursor right, to the first letter of the word following the current word. If you issue this request when the cursor is to the right of the beginning of the first word on the line, the cursor moves to the last column of the current line.

▶ **SHIFT** **STATUS**

Displays the Forms Editor status message on the system status line.

▶ **STATUS**

Displays the system message on the system status line.

▶ **TAB**

Moves the cursor to the next tab stop to the right.

- ▶ **WORD**  (or **SHIFT** )

Moves the cursor left, to the first letter of the current word or, when the cursor is at the beginning of the current word, to the first letter of the previous word. If you issue this request when the cursor is at or to the left of the beginning of the first word on the line, the cursor moves to the first column of the current line.

- ▶ **WORD**  (or **SHIFT** )

Moves the cursor right, to the first letter of the next word. If you issue this request when the cursor is to the right of the beginning of the first word on the line, the cursor moves to the last column of the current line.

- ▶ **WORD** **CHANGE CASE** 

Changes the first letter of the current word to uppercase.

- ▶ **WORD** **CHANGE CASE** 

Changes all letters of the current word to lowercase.

- ▶ **WORD** **CHANGE CASE** 

Changes all letters of the current word to uppercase.

Menu Edit Requests

To issue a menu edit request, first press the **(MENU)** key to display the request menu. Then type a letter corresponding to one of the options listed.

When you press the **(MENU)** key, the Forms Editor displays the form shown in Figure 4-3.

A En/disable request menu display	Q Quit
D Delete field	R Read field
E Enter field	S Set/modify form options
F Add/modify field	U Update fields
G Global replace	V Define/modify video display modes
I Insert window field	W Write
L Insert literal	X Show exact form
N En/disable line number mode	Z Set bell column
O En/disable overlay mode	

Figure 4-3. The Forms Editor Request Menu

Note: Some terminal types might not define a **(MENU)** key. For these terminals, each of the menu edit requests can be implemented as a direct edit request. Check with your system administrator. (See also Appendix D, "Terminal Requirements.")

This section gives an explanation of each menu request. For an online explanation of any Forms Editor request, press the **(HELP)** key while in the Forms Editor.

► **(MENU)** A En/disable request menu display

This request allows you to stop the Forms Editor from displaying the menu whenever the **(MENU)** key is pressed. Normally, the menu is displayed whenever you press the **(MENU)** key. After you have learned the contents of the menu, you might find this unnecessary. The keystrokes you use to issue menu edit requests do not change, but execution is slightly faster if the menu is not displayed. You can re-enable menu display by issuing the **(MENU)** A request again.

► **(MENU)** D Delete field

This request removes a previously defined field from the form. When you issue this request, the Forms Editor prompts you for the name of the field to delete.

► (MENU) E Enter field

This request stores a copy of a field definition so that it can be used for another field in the same form or in another form. When you issue the (MENU) E request, the cursor must be positioned to a field. The Forms Editor writes the field definition for the field to a file in the field-definitions directory. You specify the field-definitions directory with the `-library` option in the `icss_edit_form` command. The default directory is a subdirectory of your current directory named `accept_field_definitions`. The Forms Editor creates the directory, if necessary. The name of the file is the name of the field. If you specify a prefix for the form, that prefix is **not** applied to the file name. The cursor blinks once to indicate that the field definition has been saved. The form is not modified.

(MENU) E allows you to build a library of field definitions.

To retrieve a previously saved field definition, use the (MENU) R request.

► (MENU) F Add/modify field

This request allows you to define or modify a field at the current position. The Forms Editor displays a form in which you enter information about the field and its display type. The Forms Editor shows the initial or default value for each option; when modifying an existing field, the values are the current values for that field. The form is described later in this chapter under the heading “The Add/modify field Request.”

► (MENU) G Global replace

This request replaces every instance of a specified character string in the text with another. The Forms Editor prompts you for the two strings. The replacement is performed on all background text between the current position and the end of the edit buffer.

► (MENU) I Insert window field

This request allows you to define or modify a window field. The Forms Editor displays a form in which you enter the window options. The Forms Editor shows the initial or default values for the options. If you are modifying an existing window field, the values shown are the current values for that field. The form is described later in this chapter under the heading “The Insert window field Request.”

► **(MENU) L** Insert literal

This request allows you to insert characters from any supplemental character set into the form's background text. After issuing this request, you can type either the hexadecimal rank of a character from the default character set, or the hexadecimal rank of a single-shift character followed by the hexadecimal rank of a supplemental character. The specified character is inserted at the current position in the form.

For information on supplemental characters, see the subsection "International Character Set Support" in Chapter 3, "The Elements of FMS."

► **(MENU) N** En/disable line number mode

This request allows you to display line numbers at the left edge of the edit buffer. These numbers have no effect on the form. If you reissue the **(MENU) N** request, the line numbers disappear.

► **(MENU) O** En/disable overlay mode

This request allows you to change the edit mode from overlay to insert or from insert to overlay. These modes are explained under the heading "Entering Text" earlier in this chapter. At the start of each editing session, the Forms Editor is in overlay mode.

► **(MENU) Q** Quit

This request ends the editing session and returns you to command level. The Forms Editor produces no new files when **(MENU) Q** is executed; you must issue the **(MENU) W** request first to save your work. The Forms Editor prompts you before quitting if you have made changes without writing the form.

► **(MENU) R** Read field

This request creates a field at the current position, based on a previously saved field definition. The Forms Editor prompts you for the name of a field definition file. You can give either the path name of a field definition file or a simple file name. If you give a simple file name, the Forms Editor looks for that file in the field-definitions directory you specify with the `-library` option of the `icss_edit_form` command. (The default directory is a subdirectory of your current directory named `accept_field_definitions`.)

Field definition files are created by the **(MENU) E** request.

The **(MENU) R** request reads the field definition file you specify, and displays the information in the same form displayed by the **(MENU) F** option. This form is described later in this chapter under the heading, "The Add/modify field Request." You can modify the information before submitting the form.

When you submit the form, a field is created with the options and attributes specified.

► **(MENU)** S Set/modify form options

This request allows you to set or modify the form options. The Forms Editor displays a form in which you enter the options. This form shows the initial or default values for the options. If you are modifying an existing form, the values shown are the current values for that form. The form displayed by this request is explained later in this chapter under the heading “The Set/modify form options Request.”

► **(MENU)** U Update fields

This request allows you to update one or more fields in the form. The Forms Editor prompts for the name of a field. The name you specify can be a star name. To update all fields, supply an asterisk for the field name. An asterisk by itself matches any name. The Forms Editor displays a form for each field that matches the star name and lets you modify the field options and display-type options. The form is the same one displayed by the Add/modify field menu request. This form is described later in this chapter under the heading “The Add/modify field Request.”

► **(MENU)** V Define/modify video display modes

This request allows you to define or redefine the video attributes of a field or a region of background text. To specify a field, move the cursor to the field and issue the **(MENU)** v request. To specify a region of background text, first move the cursor to the beginning or end of the region and issue the **(MARK)** edit request to enable highlighting. Next, move the cursor to the other end of the region. You can further restrict the region by issuing the **(COLUMN)** edit request. When the region you wish to modify is highlighted, issue the **(MENU)** v request.

When you issue the **(MENU)** v request, the Forms Editor displays a form in which you choose the video attributes for the field or region. The Forms Editor shows the initial or default values for the attributes. If you are modifying the video attributes of an existing field, the default values are the current values for that field’s display type.

The form and the video attributes you can set are explained later in this chapter under the heading “The Define/modify video display modes Request.”

► **(MENU) W Write**

This request writes the following files:

- the form definition file *form_name.form*
- the form object module *form_name.obj*
- any field-values or field-IDs files requested in *icss_edit_form* options or with the **(MENU) S** request.

Note: The Forms Editor deletes trailing spaces from all values you enter into the editor's request forms, and from all lines entered in the form you are constructing. It also deletes all empty lines from the bottom of the form. The files written by the Forms Editor reflect these deletions.

► **(MENU) X Show exact form**

This request displays the form as it is currently defined. The Forms Editor clears the screen before constructing the form. While you are in the edit buffer, the Forms Editor fills each defined field with block graphic characters and underlines each region of the form for which you have defined modes. However, in response to this request, the Forms Editor shows the form as the Forms Processor will display it when the program runs. (One exception: field IDs appear in the fields during a **(MENU) X** display, but not at execution time.)

You can test the form's layout by entering values in the fields as a terminal user would. Return to edit request level by pressing the **(ENTER)** key, the **(CANCEL)** key, or any of the function keys defined for the form, just as a user would submit or cancel the form. The editing/function keys available while the form is displayed are the same ones available for a user when the form is displayed at execution time.

► **(MENU) Z Set bell column**

This request sets the column at which the Forms Editor sounds the terminal bell. The Forms Editor prompts you for a column number. If you type in or beyond that column during the editing session, the Forms Editor sounds the terminal bell.

The Add/modify field Request

This section describes in detail the **(MENU) F** request, Add/modify field. Use the **(MENU) F** request to create or modify any non-window field.

Note: Use the **(MENU) I** request to create or modify a **window** field.

When you issue the `(MENU) F` request, the Forms Editor displays a form containing field options and display-type options. The form is shown in Figure 4-4. Complete and submit this form to define or redefine a field and place it in your form. The display-type options you specify describe the predefined display type for the field.

Fields defined in the Forms Editor can be modified in a screen statement. See Chapter 6, “Field Descriptions” for further information. For information on changing the display-type of a field within an application, see Chapter 7, “Display Types.”

The `(MENU) F` request can be used either to modify an existing field or to create a new field. These two uses are described in the following subsections.

Modifying Existing Fields

To modify an existing field within the Forms Editor, move the cursor to the field, and issue the `(MENU) F` request. The Forms Editor displays a field options and display-type options form for you to update; the form contains the current definition of the field and its display type.

Caution: Using this request on an existing window field converts the window to an array field.

Creating New Fields

When you use the `(MENU) F` request to create a field, you must first specify the position of the field. You can do this in two ways. You can highlight the character positions in the form you want the field to occupy, or you can just position the cursor to the location where you want the field to start.

If you create a field that does not have a position within the form, or that would overlap another field, that field is *uncommitted*. Uncommitted fields are discussed later in this subsection.

Remember that every field or array element in a form must be preceded and followed by a space character on the same line of the form. However, one field's trailing space can double as the next field's leading space. The Forms Processor requires these blank positions to control the attributes of the fields and background regions on some types of terminals. A field can never begin in column 1 of a form.

You can create two types of fields with the `(MENU) F` request: simple fields and array fields. The following subsections describe how to define these two types of fields.

Simple Fields. A *simple field* is a single non-array field. A simple field takes up one or more columns on one row of a form.

If you want to specify the position of a simple field by highlighting a region of text, first position to one end of the region. Set a mark with the `(MARK)` edit request,

and move the cursor to the other end of the region. The highlighted characters appear in reverse video. (Note that if you highlight a region of empty space, the highlighting is not visible, as it is when characters are highlighted.) The entire region must be on a single line. Next, issue the **(MENU)** F request. The field-options form appears with the **POSITION** and **LENGTH** values already filled in. You can alter the field by changing the displayed values.

Alternatively, you can specify the field position by just moving the cursor to the location where you want the field to start and issuing the **(MENU)** F request. In this case, you must supply a **LENGTH** value in the field-options form.

Array Fields. You can use highlighting to mark the location of an array field only if the array elements are arranged in a single column in the form, with no blank lines between the elements. First, position to one corner of the array and issue the **(MARK)** edit request. Next, position to the opposite (diagonal) corner of the array and issue the **(COLUMN)** edit request. The region where the array will be located is now highlighted. Issue the **(MENU)** F request. The Forms Editor derives the field's **LENGTH**, **POSITION**, and **ARRAY LAYOUT** values from the highlighted region.

Alternatively, you can specify the location of the array by placing the cursor at the top left corner of the array before typing **(MENU)** F. In this case, the field's **POSITION** value appears in the field-options form, but its **LENGTH** and **ARRAY LAYOUT** values do not. Note that you must use this method for multi-column arrays or for arrays that use blank lines to separate elements.

See the array field option in Chapter 6, "Field Descriptions," for more information on array fields.

Uncommitted Fields. If you define a field without specifying a position (by deleting the values in the **POSITION** option of the **(MENU)** F form), that field is uncommitted. Similarly, if you define a field that partially or completely overlaps an existing field, the Forms Editor rejects the field's position value, leaving the field uncommitted.

An uncommitted field is associated with the form but does not appear in it. However, the Forms Editor assigns a field ID and any other field options you specify to the field and writes the field to the form definition file. The **DISABLE_ENTIRE_FIELD** options form option switch is initially true for an uncommitted field. A screen statement can commit the field by specifying a location with the **position** field option and setting the **DISABLE_ENTIRE_FIELD** switch to false. You can also commit a field by choosing a position for it during any editing session.

Within the Forms Editor, you must use the **(MENU)** U request to reference an uncommitted field.

The Add/modify field Form

When you invoke the **(MENU) F** option, the form in Figure 4-4 appears on your screen.

Note: Some fields of the **(MENU) F** form appear only when certain form options are enabled. See the description of each field for information.

Field Options			
FIELD NAME	<input type="text"/>		
POSITION	<input type="text"/> 1	,	<input type="text"/> 2
ARRAY LAYOUT	<input type="text"/>	,	<input type="text"/>
LENGTH	<input type="text"/>	BASIC	\$= <input type="text"/>
FIELD TYPE	input	COBOL display	<input type="text"/>
DISABLE	no	FORTTRAN character*	<input type="text"/>
REQUIRED	no	PASCAL char array	<input type="text"/>
DISAPPEARING	no	PL/1 char / pic	<input type="text"/>
IN FIELD-VALUES	yes	C char []	<input type="text"/>
SHIFT, DCS =	none	field-values sequence	<input type="text"/> 0
INITIAL	<input type="text"/>		
HELP	<input type="text"/>		
Displaytype Options			
DISPLAYTYPE NAME	<input type="text"/>		
PICTURE	<input type="text"/>		
VALUE RESTRICTION	none	VALIDATE	<input type="text"/>
INTENSITY	high	underline	<input type="text"/> not inverse
		non blinking	<input type="text"/> not blanked
JUSTIFICATION		TRIM BLANKS	<input type="text"/> yes CHAR SET <input type="text"/> any
AUTO TAB	no	BANK TELLER DECIMAL	<input type="text"/> no INDEXED CYCLE LIST <input type="text"/> no
TRAP ON FIELD EXIT	no	TRAP ON FIELD ENTRY	<input type="text"/> no
FORCE INSERT MODE	no	FORCE OVERLAY MODE	<input type="text"/> no

Figure 4-4. Form Displayed by the **(MENU) F (Add/modify field) Request**

The following subsection describes the field options part of Figure 4-4. The subsequent subsection describes the display-type options part.

The Field Options

Table 4-2 lists the field options from the **(MENU)** F field options form and their screen statement counterparts. The field option descriptions in Chapter 6, "Field Descriptions," contain more detailed information on these options.

Table 4-2. The Forms Editor Field Options

Forms Editor	Field Description
ARRAY LAYOUT _____, _____ ROW SPACING _____, COLUMN SPACING _____ 1	array (number_rows, number_columns)
DISABLE (no, yes)	datastate (data_state_switches)
DISAPPEARING (no, yes)	datastate (data_state_switches)
FIELD NAME _____	field (field_id)
FIELD TYPE (input, output, output only)	update (field_value_variable) datastate (data_state_switches)
field-values sequence	N/A
HELP _____	help (help_message)
IN FIELD-VALUES (yes, no)	N/A
INITIAL _____	initial (initial_value) or initial_value_string
LENGTH _____	length (field_length)
POSITION _____, _____	position (line, column)
REQUIRED (no, yes)	datastate (data_state_switches)
SHIFT, DCS = (none, latin 1, kanji, katakana, hangul) [†]	shift (character_set_id)
UNSHIFT, RGS = (none, latin 1, kanji, katakana, hangul) [†]	unshift (character_set_id)
BASIC (\$=, \$<=, #=, %=31, %=15, =15, =6)	N/A
COBOL (display, display-2, comp-6, comp-5, comp-4, comp-2, comp-1, comp-3)	N/A
FORTTRAN (character*, string*, integer*4, integer*2, real*8, real*4)	N/A
PASCAL (char array, string, integer, -32K..32K, real)	N/A
PL/1 (char / pic, char var, fixed dec, fixed(31), fixed(15), float(53), float(24))	N/A
C (char [], char_var'ng, int, short, double, float, uns'd short, unsigned)	N/A

[†] The title of the SHIFT, DCS = field is itself a two-value cycle field. The second value is UNSHIFT, RGS =.

The remainder of this subsection describes the field options that appear in the (MENU) F form. They are described in the same order in which they appear in the form, except that the language data-type fields are grouped together and described at the end.

Where appropriate, cycle list values are written to the right of the option name. The first value in a cycle list is the default value.

► FIELD NAME

This option specifies the name of the field. Every field must have a name. When you create a new field, the Forms Editor derives an initial field name from any background text to the right of the field. When you are modifying an existing field, the Forms Editor initially displays the current name of the field. You can rename an existing field by changing the name displayed in the FIELD NAME option.

► POSITION

This option specifies the starting row and column of the field. If you are modifying an existing field, the field's current starting row and column are displayed. If you are defining a new field and have highlighted the region that the field will fill, the initial values are the row and column of the start of the field. If you are defining a new field and have not highlighted a region, the initial values are the current cursor position. You can change the initial values. (Remember that a field can never begin in column 1, because an empty space is required at the beginning and end of every form field.)

► ARRAY LAYOUT

This option allows you to create an array field by specifying the number of rows and columns in the array. (The LENGTH option in the (MENU) F form specifies the length of each element of the array.) See the description of the array field option in Chapter 6, "Field Descriptions."

► ROW SPACING

This option specifies the number of blank rows between rows of elements in an array field. The default is 0.

► COLUMN SPACING

This option specifies the number of blank columns between columns of elements in an array field. The default is 1. The value must be at least 1, because a blank space is required before and after every form field or array element.

► **LENGTH**

This option specifies the length of the field, in character positions. For an array field, the **LENGTH** option specifies the length of each array element. You must specify a positive value for this option.

► **FIELD TYPE** (CYCLE) input, output, output only

This option specifies whether the user can position to and type in the field. The user can position to (and usually type in) input fields; the user cannot position to output or output-only fields. For any of the three field types, the Forms Editor declares a data-structure component that corresponds to the form field in any field-values file it creates.

The default video attributes of input fields differ from that of output-only fields. If you select the type output, the field is given the attributes of an output-only field, but the application program can change the field to an input field by setting the **INPUT_FIELD** data-state switch to true. Output-only fields cannot be changed to input fields by the application.

The distinction between the field types output and output only is relevant only if a field-values file is generated.

For more information, see the description of the **INPUT_FIELD** switch in Chapter 8, "Data States." See also the information under the heading "Input Fields and Output Fields" in Chapter 3, "The Elements of FMS."

► **DISABLE** (CYCLE) no, yes

This option allows you to initially disable the entire field. If you set **DISABLE** to yes, the **DISABLE_ENTIRE_FIELD** data-state switch is set to true.

For more information, see the description of the **DISABLE_ENTIRE_FIELD** switch in Chapter 8, "Data States."

► **REQUIRED** (CYCLE) no, yes

This option allows you to specify that a value must be given for the field before the form is submitted. If you specify yes, the **REQUIRED_FIELD** data-state switch is initially true for the field. For more information, see the description of the **REQUIRED_FIELD** switch in Chapter 8, "Data States."

By default, required fields appear in reverse video in a form. You can change the default appearance of required fields by using the (MENU) S (Set/modify form options) request.

► DISAPPEARING ☐ no, yes

This option allows you to specify that the output field value disappears when the user begins typing in the field. If you specify yes, the DISAPPEARING_DEFAULT data-state switch is initially true for the field.

For further information, see the description of the DISAPPEARING_DEFAULT switch in Chapter 8, “Data States.”

► IN FIELD-VALUES ☐ yes, no

This option allows you to specify whether a variable declaration for the field should be included in the field-values file. (An entry for the field is included in any field-IDs file regardless of the value of this option.)

One use for this option is to effectively create an area of background text that can have its visual attributes changed by a program. To do this, create an output field with the IN FIELD-VALUES option set to yes, and set its initial value to the appropriate background text.

► $\left\{ \begin{array}{l} \text{SHIFT, DCS =} \\ \text{UNSHIFT, RGS =} \end{array} \right\}$ ☐ none, latin 1, kanji, katakana, hangul

The title for this option is itself a cycle field. The title SHIFT, DCS = enables the shift field option with the character set specified. In this case, the field value is the default character set for the field (the only character set that is not to be shifted). The title UNSHIFT, RGS = enables the unshift field option. In this case, the field value specifies the only right graphic set that the field supports.

This option does not appear in the Field Options form until either the INTO form option or the -into command line option has been specified along with one or more language options.

The shift and unshift field options are described in Chapter 6, “Field Descriptions.”

► field-values sequence

This option allows you to specify the position of the field's corresponding declaration in any field-values include files that the Forms Editor constructs for the form. The default value for the sequence number is zero.

The field-values sequence option does not appear in the (MENU) F form until either the INTO form option or the -into command-line option has been specified along with one or more language options.

The Forms Editor determines the order of the declarations in the field-values file by sorting the fields by the specified sequence number: lowest numbers sort first. If two or more fields have the same field-values sequence number (as in the default case, where all fields have a sequence number of zero), then the Forms Editor sorts those fields according to the cursor order of the fields (left-to-right in the first row, then left-to-right in the second row, etc.) The sequence number can be any positive or negative integer.

► INITIAL

This option allows you to specify an initial output value for the field. See Chapter 3, "The Elements of FMS," for information on initial output values.

► HELP

This option provides a one-line message or instruction for the user. The message appears on the bottom line of the screen when the user moves the cursor into the field and presses the (HELP) key.

► BASIC COBOL FORTRAN PASCAL PL/1 C (CYCLE) (See Table 4-3)

These options allow you to select the data type of the variable or data-structure member that will hold the field's value. Table 4-3 shows the possible data types for each language.

The Forms Editor only enables and displays the fields of the programming languages for which it is constructing field-values include files. The language data-type fields appear in the (MENU) F form only if you have selected one or more languages with either the command-line options or the (MENU) S form options, and specified either the -into command-line option or set the PRODUCE INTO field of the (MENU) S form to yes. Furthermore, no language data-type field appears in the form until a LENGTH value is supplied.

When constructing include files for more than one language, the data types selected must be consistent. If you do not specify any data type for the field, the Forms Editor assigns a default data type to the field. The default data type is derived by the following rules:

1. If the UNSHIFT option is used for the field, then the default data type is a fixed-length character string with the same length as the field.
2. If Rule 1 does not apply and a field-values file is being created for FORTRAN and the (MENU) S FORTRAN STRINGS field is set to no, then the default data type is a fixed-length character string with twice the length of the field.
3. If Rules 1 and 2 do not apply, then the default data type is a varying-length character string with a maximum length of twice the length of the field.

The string data types derived by Rules 2 and 3 are twice as long as the field to allow space for single-shift characters within the string.

When a language and the PRODUCE INTO option are specified after fields have already been defined, the field input variables are assigned the default data type. You can change the data type by modifying the fields with either the (MENU) F or (MENU) U option.

In addition to the data-type cycle field, each language has a second field for auxiliary information on the data type. Table 4-4 shows the auxiliary information required for some data types in each language.

Table 4-3. Data Types for Field-Values Variable Declarations

BASIC	COBOL	FORTTRAN	Pascal	PL/I	C
\$=	display	character*	char array	char / pic	char[]
\$<=	display-2	string*	string	char var	char_var'ng
#=	comp-6			fixed dec	
%=31	comp-5	integer*4	integer	fixed(31)	int
%=15	comp-4	integer*2	-32K..32K	fixed(15)	short
=15	comp-2	real*8	real	float(53)	double
=6	comp-1	real*4		float(24)	float
					uns'd short
	comp-3				unsigned

Table 4-4. Auxiliary Information for Data Types

Language	Data Type	Auxiliary Information
BASIC	\$= \$<= # =	<i>number_of_characters</i> <i>max_number_of_characters</i> <i>precision, scale</i>
C	char[] char_var'ng	<i>number_of_characters</i> <i>max_number_of_characters</i>
COBOL	display display-2 comp-3 comp-6	<i>picture</i> <i>pic x(max_number_of_characters)</i> <i>numeric_picture</i> <i>numeric_picture</i>
FORTRAN	character* string*	<i>number_of_characters</i> <i>max_number_of_characters</i>
PASCAL	char array string	<i>number_of_characters</i> <i>max_number_of_characters</i>
PL/1	char / pic char var fixed dec	<i>number_of_characters</i> <i>'picture'</i> <i>max_number_of_characters</i> <i>precision, scale</i>

Note: The FORTRAN string data type is converted to a common block if the `-no_fortran_strings` command-line option is given and the FORTRAN STRINGS **(MENU)** S option is set to no.

In Table 4-4, the values *number_of_characters* and *max_number_of_characters* must be positive integers. The *precision* value must be a positive integer, and *scale* must be an integer. The *precision* is the maximum number of decimal digits allowed in the value. The *scale* gives the implicit location of the decimal point. Thus, a positive *scale* is the number of digits in the fractional part, and a negative *scale* represents the number of omitted zeros in the value entered by the user.

Display-Type Options

Table 4-5 lists the display-type options from the **(MENU)** F form and their screen statement counterparts. The display-type option descriptions in Chapter 7, “Display Types,” contain more detailed information on these options.

Table 4-5. The Forms Editor Display-Type Options

Forms Editor	Display-Type Description
AUTO TAB (no, yes)	action (<i>action_switches</i>)
BANK TELLER DECIMAL (no, yes)	action (<i>action_switches</i>)
CHAR SET (any, ascii, latin 1, kanji, katakana, hangul)	charset (<i>char_set_id</i>)
DISPLAYTYPE NAME	N/A
FORCE INSERT MODE (no, yes)	action (<i>action_switches</i>)
FORCE OVERLAY MODE (no, yes)	action (<i>action_switches</i>)
INDEXED CYCLE LIST (no, yes)	action (<i>action_switches</i>)
INTENSITY (high, normal, low) (underline, no underline) (not inverse, inverse) (non blinking, blinking) (not blanked, blanked)	visual (<i>visual_switches</i>)
JUSTIFICATION (, left, right, center)	visual (<i>visual_switches</i>)
PICTURE _____	picture (<i>picture</i>)
TRAP ON FIELD ENTRY (no, yes)	action (<i>action_switches</i>)
TRAP ON FIELD EXIT (no, yes)	action (<i>action_switches</i>)
TRIM BLANKS (yes, no)	action (<i>action_switches</i>)
VALUE RESTRICTION (none, range, cycle)	cycle (<i>value</i> [, <i>value</i>] ...) range (<i>low</i> , <i>high</i>)
VALIDATE _____	validate (<i>validation_entry</i>)

The remainder of this subsection describes the display-type options that appear in the **(MENU)** F form. They are described in the same order in which they appear in the form.

Where appropriate, cycle list values are written to the right of the option name. The first value in a cycle list is the default value.

► **DISPLAYTYPE NAME**

This option is currently disabled.

► **PICTURE**

This option allows you to specify a picture that restricts the values the user can type in the field. The value you specify is used as the initial value for the picture display-type option.

For further information on field pictures, see Chapter 9, "Field Pictures and Filtering." See also the description of the picture display-type option in Chapter 7, "Display Types."

► **VALUE RESTRICTION** `(CYCLE)` none, range, cycle

This option enables you to restrict the allowed values of an input field.

If you select *range*, the Forms Editor displays a form in which you enter the low and high limits of a range of values that a user can enter in the field. See the description of the range display-type option in Chapter 7, "Display Types."

If you select *cycle*, the Forms Editor displays a form in which you enter a list of up to 20 possible values for the field. For information on cycle fields, see the section "Cycle Display Types" and the description of the cycle display-type option in Chapter 7, "Display Types."

Note: If you specify an initial value for a field, it must satisfy any range or cycle restriction. A form with an invalid initial value might cause the application program to fail.

► **VALIDATE**

This option allows you to specify the name of a validation procedure, subroutine, or subprogram that checks the value the user types in an input field. The validation routine can be written in any language. It must be bound with any program that displays the form. The Forms Processor calls a validation routine to check the value after the form is submitted. If you leave this option blank, the Forms Processor does not call any validation routine for the particular input field. For more information, see the description of the validate display-type option in Chapter 7, "Display Types." For information on coding a validation routine, see Chapter 10, "Error Handling and Field Validation."

► INTENSITY

The five cycle fields in the middle of the display-type options form specify the video display attributes of the field. They specify initial values for display-type visual switches. For non-cycle input fields, the default video display is high intensity and underlined. For cycle fields, the default video display is high intensity with no underline. For output and output-only fields, the default video display is low intensity with no underline.

The default attributes for a non-cycle input field are initially shown on the (MENU) F form. If you submit the form for an output, output-only, or cycle field without changing the video attributes, the form is redisplayed with the appropriate default attributes. You can change the attributes for any field.

For more information on the video attributes, see the description of the visual display-type option in Chapter 7, "Display Types."

► JUSTIFICATION (CYCLE) null, left, right, center

This option allows you to specify the justification for the field. It initializes the LEFT_JUSTIFY_FIELD_DATA, RIGHT_JUSTIFY_FIELD_DATA, and CENTER_FIELD_DATA visual switches for the display type. At most, one of these switches can be true. You can set a switch to true by cycling to the value left, right, or center. If you set the option to null, all three switches are false.

If all three switches are false, the Forms Processor uses the default justification for the field.

See the discussion of justification under the heading "Numeric and Alphanumeric Fields" in Chapter 3, "The Elements of FMS," and the description of the visual display-type option in Chapter 7, "Display Types."

► TRIM BLANKS (CYCLE) yes, no

This option allows you to initialize the NOTRIM_FIELD_SPACES display-type visual switch. If you specify yes for the TRIM BLANKS option, then the NOTRIM_FIELD_SPACES switch is false and normal trimming occurs. If you specify no for the TRIM BLANKS option, then the NOTRIM_FIELD_SPACES switch is true and trimming is inhibited. For more information, see the description of the visual display-type option in Chapter 7, "Display Types."

- **CHAR SET** (CYCLE) any, ascii, latin 1, kanji, katakana, hangul

This option allows you to specify an initial value for the charset display-type option. This option specifies which character set the field supports. The default value, any, puts no restrictions on the field. If you specify a character set for the CHAR SET option, only characters from that set are allowed in the field.

The valid character sets for a field can also be restricted by the unshift field option or by limitations of the device on which the form is displayed.

- **AUTO TAB** (CYCLE) no, yes

This option allows you to initialize the AUTO_TAB_TO_NEXT_FIELD display-type action switch. If this switch is true for a field's display type, then as soon as the user has filled the field, the cursor moves automatically to the next field.

For more information, see the discussion of the display-type action option in Chapter 7, "Display Types."

- **BANK TELLER DECIMAL** (CYCLE) no, yes

This option allows you to initialize the BANK_TELLER_DECIMAL display-type action switch. If this switch is true for an input field with a numeric picture containing a decimal point, the field acts in the way common to bank-teller terminals: the cursor remains at the field's right boundary, and typed numerals move left without regard for the decimal point. For example, if you type 123 in a field with the display picture z.99, the display is initially .00, then .01, then .12, and finally 1.23. (The digits 0-9 are the only valid input characters in a bank-teller decimal numeric field.)

If the BANK_TELLER_DECIMAL switch is false, the cursor remains at the decimal point until the decimal point is typed; typed digits move left of the decimal. Digits typed after the decimal point appear to the right of the decimal.

You can specify the default value for the BANK TELLER DECIMAL option in the (MENU) s form.

For more information on numeric fields, see Chapter 3, "The Elements of FMS." See also the information on the BANK_TELLER_DECIMAL switch in the description of the action display-type option in Chapter 7, "Display Types."

► INDEXED CYCLE LIST ☒ no, yes

This option allows you to specify the initial setting of the INDEXED_CYCLE_LIST display-type action switch. If this switch is true for a cycle field, the value returned to the field-value variable is an integer index into the cycle list, rather than a value from the cycle list. If you use this option and the cycle list contains only two values, the field-value variable can be either an integer or a bit (1) aligned variable. If the cycle list contains more than two values, the field-value variable must be an integer.

For more information, see the discussion under the heading “Indexed Cycle Lists” and the description of the action display type option in Chapter 7, “Display Types.”

► TRAP ON FIELD EXIT ☒ no, yes

This option allows you to initialize the TRAP_ON_FIELD_EXIT display-type action switch. If this switch is true, control returns to the application program as soon as the user moves the cursor out of the field. This allows a program to check the user’s input before redisplaying the form and allowing further input.

If control returns to the application because of a trap on field exit, the value -2 is returned in the keyed form option.

For more information on the TRAP_ON_FIELD_EXIT switch, see the description of the action option in Chapter 7, “Display Types.”

► TRAP ON FIELD ENTRY ☒ no, yes

This option allows you to initialize the TRAP_ON_FIELD_ENTRY display-type action switch. If this switch is true, control returns to the application program as soon as the user positions to the field. This allows a program to check the user’s input or compute initial output values before redisplaying the form and allowing further input.

If control returns to the application because of a trap on field entry, the value -9 is returned in the keyed form option.

For more information on the TRAP_ON_FIELD_ENTRY switch, see the description of the action option in Chapter 7, “Display Types.”

► **FORCE INSERT MODE** (CYCLE) no, yes

This option allows you to initialize the FORCE_INSERT_MODE display-type action switch. If this switch is true, the initial edit mode for the field is insert mode. If both this switch and the FORCE_OVERLAY_MODE switch are false, the default edit mode is overlay if the field is left-justified, and insert if the field is right-justified. However, field justification and initial edit modes are very device dependent.

The user can change the edit mode with the (INSERT/OVERLAY) key.

For more information, see the description of the FORCE_INSERT_MODE switch in the description of the action display-type option in Chapter 7, "Display Types." For information on field justification, see the discussion under the heading "Numeric and Alphanumeric Fields" in Chapter 3, "The Elements of FMS."

► **FORCE OVERLAY MODE** (CYCLE) no, yes

This option allows you to initialize the FORCE_OVERLAY_MODE display-type action switch. If this switch is true, the initial edit mode for the field is overlay mode. If both this switch and the FORCE_INSERT_MODE switch are false, the default edit mode is overlay if the field is left-justified, and insert if the field is right-justified. However, field justification and initial edit modes are very device dependent.

The user can change the edit mode with the (INSERT/OVERLAY) key.

For more information, see the description of the FORCE_OVERLAY_MODE switch in the description of the action display-type option in Chapter 7, "Display Types." For information on field justification, see the discussion under the heading "Numeric and Alphanumeric Fields" in Chapter 3, "The Elements of FMS."

The Insert window field Request

This section describes the (MENU) I request, Insert window field.

Use (MENU) I to define a window field and insert it in your form. A *window field* is a rectangular region of a form in which a second form can be displayed and managed by the Forms Processor. A form containing a window field is called a *master form*; a form displayed in a window field is called a *subform*.

For more information on windows and subforms, see Chapter 11, "Windows and Subforms." See also the description of the window field option in Chapter 6, "Field Descriptions."

When you issue the **(MENU)** I request, the form shown in Figure 4-5 appears on your screen.

If you are creating a new window, you must indicate the position of the window before you issue the **(MENU)** I request. You indicate the position in two ways: by highlighting the entire region the window will occupy, or by positioning to the top left corner of the region.

To highlight a window region, first position to one corner of the region, and set a mark by issuing the **(MARK)** edit request. Next, move the cursor to the opposite (diagonal) corner of the region, and issue the **(COLUMN)** request. The region the window will occupy is now highlighted. Issue the **(MENU)** I request to finish defining the window. When you highlight a region before issuing the **(MENU)** I request, the Forms Editor derives the **POSITION** and **SUB-FORM SIZE** values from the highlighted region. You can change these values if you wish.

If you just position to the top left corner of a region and issue the **MENU** I request, the Forms Editor derives the **POSITION** values from the cursor position, but you must supply the **SUB-FORM SIZE**.

As with all other fields, you must precede and follow every line of a window field with a space character. For a window with more than one row, you must allow for a column of space characters both to the left and to the right of the window. The Forms Processor positions a subform in the master form so that the subform's leading column of attribute characters overlaps the window field's column of attribute characters.

-- Window Field Options --

FIELD NAME	<div style="background-color: black; width: 150px; height: 1.2em;"></div>
POSITION	<div style="border-bottom: 1px solid black; width: 40px; display: inline-block;"></div> / <div style="border-bottom: 1px solid black; width: 40px; display: inline-block;"></div>
SUB-FORM SIZE	<div style="border-bottom: 1px solid black; width: 40px; display: inline-block;"></div> / <div style="border-bottom: 1px solid black; width: 40px; display: inline-block;"></div>

Figure 4-5. Form Displayed by the (MENU) I (Insert window field) Request

The following are descriptions of the fields in the **(MENU)** I form.

► FIELD NAME

This option specifies the name of the window field. If you are modifying an existing field, its current name is the default value.

► POSITION

This option specifies the row and the column of the left corner of the window. Note that the row number is specified first.

► SUB-FORM SIZE

This option specifies the number of lines (or rows) in the window followed by the number of characters (or columns) in each line.

The Set/modify form options Request

This section describes the `(MENU) S` request, Set/modify form options.

Use the `(MENU) S` request to specify the form options of a form.

When you type `(MENU) S`, the Forms Editor displays the form shown in Figure 4-6.

```

-- Form Options -- for form_name

PREFIX
MASKKEYS      no
PRODUCE INTO  no
BASIC         no
COBOL         no
FORTRAN       no  STRINGS yes
PASCAL        no
PL/1          no
C             no
VALIDATE      and report errors
WIDE CURSOR   no
MESSAGE
CURSOR FIELD  _____ INDEX 1
ERROR MESSAGE FIELD

BACKGROUND MODE
  INTENSITY:   low      no underline    not inverse
                  non blinking    not blanked

REQ FIELD MODE TOGGLES
  INTENSITY TOGGLE: same intensity  same underlining  toggle inverse
                  same blinking     same blanking

```

Figure 4-6. Form Displayed by the `(MENU) S` (Set/modify form options) Request

The rest of this section describes the options shown in Figure 4-6. The options are described in the order they appear in the form, except that the language options and the `STRINGS` option are grouped together and described at the end.

► *form_name*

The name of the form is displayed on the top line of the form. You can position to the name and change it. The form name is used to name the files produced by the Forms Editor.

► **PREFIX**

This option allows you to specify a prefix for the field-ID constant names in the field-IDs files. If you specified the `-prefix` option in the `icss_edit_form` command, the form name followed by an underline is the default value for this option. Otherwise, the default is not to use a prefix. Any prefix value you supply in the Form Options form replaces any prior value.

You should use a prefix unless you plan to use only one form in any application that calls the form. If an application calls two or more forms with common field names and no **PREFIX** value, the respective field-IDs include files will have conflicting `%replace` statements for the common field names.

See also the description of the `-prefix` argument of the `icss_edit_form` command earlier in this chapter.

► **MASKKEYS** **(CYCLE)** no, yes

This option allows you to enable function keys other than **(ENTER)** and **(CANCEL)** that a user can use to submit or cancel the form. The program can determine what key was used to submit or cancel the form and can then choose a course of action based on that information. A code specifying the key used is returned by the keyed form option. Masked keys are commonly used in menu forms.

If you have set the **MASKKEYS** option to yes, a mask-keys form appears when you submit the Form Options form. This form is shown in Figure 4-7.

Define Key Mask

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
B	B	B	B	B	B	B	B	B	B	B	B	B	B	B	B

17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32
B	B	B	B	B	B	B	B	B	B	B	B	B	B	B	B

E <=> "enter", B <=> "beep" and C <=> "cancel"

Figure 4-7. The Forms Editor Mask-Keys Form

The mask-keys form contains cycle fields for 32 function keys. For each function key, you define the action of the key: E (enter) means a user can submit a form by pressing the key, C (cancel) means a user can cancel a form by pressing the key, and B (beep) means the key has no function for this form and the terminal bell sounds when the key is pressed. The default is B for all function keys.

Note: Most terminals have fewer than 32 available function keys; **MASKKEYS** settings for unusable function keys are ignored.

For more information, see the description of the **maskkeys** form option in Chapter 5, "Form Options."

► BANK TELLER DECIMAL (CYCLE) no, yes

This option allows you to set the default for the BANK TELLER DECIMAL option of the (MENU) F (Add/modify field) form. For information on the BANK TELLER DECIMAL option of the (MENU) F form, see the discussion of that form earlier in this chapter.

► PRODUCE INTO (CYCLE) no, yes

This option allows you to specify whether you want the Forms Editor to generate field-values files for the languages specified in the (MENU) s form. If you specify yes for PRODUCE INTO, the Forms Editor produces field-value files for each of the languages specified. If you specify no for PRODUCE INTO, no field-value files are produced. The PRODUCE INTO and (MENU) s language options override the equivalent -into and language options of the icss_edit_form command. For information on the command-line options, see the description of the icss_edit_form command earlier in this chapter.

► decimal is period (CYCLE) decimal is period, decimal is comma

This option allows you to choose the decimal-point character for numeric fields. If you specify decimal is period, the period character acts as a decimal point and the comma character is used to group digits. If you specify decimal is comma, the meanings of these two characters are reversed.

For more information on decimal points and grouping characters in numeric fields, see Chapter 9, “Field Pictures and Filtering.”

► CURRENCY SYMBOL

This option allows you to specify a currency symbol. The currency symbol is filtered from field values before they are loaded into numeric variables, and from program-generated output values before they appear in form fields.

The default currency symbol is the dollar sign (\$).

For more information on filtering field values, see Chapter 9, “Field Pictures and Filtering.”

► INITIAL DISPLAY (CYCLE) clear, scroll

This option allows you to specify how the form is initially displayed. If INITIAL DISPLAY is set to clear, the Forms Processor clears the user's screen and displays the form at the top of the screen. If you choose scroll, the Forms Processor scrolls up text on the user's screen and displays the form below the text. See also the description of the origin form option in Chapter 5, “Form Options.”

► ALTERABLE BY ACCEPT (CYCLE) yes, no

This option allows you to specify that certain attributes of the form cannot be changed in an accept or perform screen initialization statement. If a form is **not** ALTERABLE BY ACCEPT, values supplied in the Forms Editor for the form and field options listed in Table 4-6 cannot be changed by an accept or perform screen initialization statement. A statement option that would normally override the Forms Editor value is ignored when ALTERABLE BY ACCEPT is set to no **and** a value is given for the option in the Forms Editor.

Table 4-6. Form and Field Options Affected by ALTERABLE BY ACCEPT

Form Options (MENU) S)	Field and Display-Type Options (MENU) F)
BEEP MASKKEYS MESSAGE	AUTO TAB <i>cycle list values</i> DISAPPEARING FIELD TYPE FORCE INSERT MODE FORCE OVERLAY MODE HELP INITIAL PICTURE <i>range limits</i> TRAP ON FIELD ENTRY TRAP ON FIELD EXIT VALIDATE The five video attribute options

The preset values for all of these options **can** be changed in a perform screen initialization statement if ALTERABLE BY ACCEPT is set to yes, the default value. Furthermore, if no value is supplied for the option in the Forms Editor, a value can be supplied in an accept or perform screen initialization statement, whether the form is ALTERABLE BY ACCEPT or not.

Note that the ALTERABLE BY ACCEPT option does **not** inhibit changes made in perform screen update, perform screen input, or perform screen output statements.

► BEEP ☐ no, yes

This option specifies whether the terminal bell sounds every time the Forms Processor displays or redisplay the form. The default is no. If you select no, the terminal bell sounds only if you specify so in a beep form option within a screen statement. If you select yes, the Forms Processor disregards a beep form option in a screen statement, and the terminal bell sounds each time the form is displayed.

► TIME OUT (sec) + (1/1024ths)

These two fields allow you to specify a maximum period of time to wait for user input for a perform screen input statement. These fields specify the initial value for the timeout form option.

In the ☐ s options, you can specify the timeout period as an integral number of seconds and an integral number of 1024ths of a second. For example, to set a timeout period of four and one-half seconds, specify 4 in the TIME OUT (sec) field and 512 in the + (1/1024ths) field.

The default value of -1 does not set any timeout period for the form. If no timeout period is specified, the Forms Processor waits indefinitely for user input.

For further information, see the description of the timeout form option in Chapter 5, "Form Options."

► VALIDATE ☐ and report errors, and ignore errors, field under cursor

This option allows you to initialize the `VALIDATE_ERRORS_OFF` and `VALIDATE_ONE_FIELD` switches of the options form option. These switches indicate how field validation routines are treated when a trap on field exit or trap on field entry occurs. If you specify and report errors, both switches are set to false. This means that when a trap on field exit occurs, all field validation routines are invoked and errors are reported.

If you specify and ignore errors, the `VALIDATE_ERRORS_OFF` switch is set to true. This means that when a trap on field exit occurs, all validation routines are invoked, but validation errors are reported only for the trap field.

If you specify field under cursor, the `VALIDATE_ONE_FIELD` switch is set to true. This means that when a trap on field exit occurs, only the validation routine for the trap field is invoked; validation routines for other fields are ignored.

For more information, see the description of the options form option in Chapter 5, "Form Options."

► CHECK 3270 MODEL ☐ no, yes

This option allows you to ensure that the form you define will work on a 3270 terminal. If you specify yes for this option, then when you write the form with the ☐ w request, the Forms Editor checks for form options or display-type action options that make the form unusable on a 3270. If such a feature is in use, the Forms Editor returns one of the following messages:

This form OPTION does not work at all on a 3270.

This displaytype ACTION does not work at all on a 3270.

In addition, the Forms Editor does **not** write any field-values files, field-IDs files, or a form object module.

Note that only features that significantly affect form processing are flagged. Such features include the VERTICAL SCROLL TRAP ☐ s option, and the TRAP ON FIELD ENTRY and TRAP ON FIELD EXIT ☐ F options. Convenience features, such as the REQUIRED and DISAPPEARING form options, are not flagged.

The CHECK 3270 MODEL option also initializes the CHECK_3270_FORMS_MODEL options form option switch. If this switch is true, the Forms Processor checks at application runtime for features unsupported by 3270 terminals. For information on this switch, see the description of the options form option in Chapter 5, "Form Options."

► WIDE CURSOR ☐ no, yes

This option allows you to initialize the WIDE_CURSOR switch in the options form option. If this switch is true, the user cannot type in input fields. When the cursor is positioned to an input field, the entire field is highlighted. The user can position to a particular field and then submit the form. The application can determine which field the user chose by using the getcursor option.

For more information on the WIDE_CURSOR switch, see the description of the options form option in Chapter 5, "Form Options."

► VERTICAL SCROLL TRAP ☐ no, yes

This option allows you to initialize the VERTICAL_SCROLL_TRAP switch in the options form option. If this switch is true, when the user tries to move beyond the last field in the form or before the first field, control returns to the application.

For more information on the VERTICAL_SCROLL_TRAP switch, see the description of the options form option in Chapter 5, "Form Options."

► MESSAGE

This option allows you to specify an initial value for the message form option. When the form is displayed, the message you supply is displayed on the terminal's bottom line.

► CURSOR FIELD, INDEX

These two options allow you to specify the initial cursor position for the form. In the CURSOR FIELD option, you can specify the name of the field to which you want to position the cursor. In the INDEX option, you can specify the cursor's character position within that field. If INDEX is 1 (the default), the cursor is positioned to the first character position in the field; if INDEX is 2, the cursor is positioned to the second character position in the field, and so forth.

The INDEX value is ignored for some devices.

The putcursor form option overrides these options. For more information, see the description of the putcursor option in Chapter 5, "Form Options."

► ERROR MESSAGE FIELD

This option is currently disabled.

► BACKGROUND MODE

This set of options allows you to choose the default video display attributes for background text in the form. You can change the video display attributes for specific text by using the (MENU) v option. If you want the ability to change video attributes in the application, put the text in an output-only field.

► REQ FIELD MODE TOGGLES

This set of switches describes how required fields should be highlighted when the field contains its null value. Highlighting is accomplished by changing — or toggling — one or more video attributes of the field.

These five cycle fields indicate whether each video attribute should be changed from the field definition (toggle) or left the same as in the field definition (same) for all required fields. With the INTENSITY TOGGLE, you can choose to toggle the intensity of low-intensity fields only, high-intensity fields only, or both low- and high-intensity required fields by choosing the cycle values low, high, or low & high, respectively.

By default, the inverse attribute chosen for every required field is reversed, while all other video display modes are left as defined in the respective field definitions.

As soon as the user types a non-null value in the field, the video attributes return to their normal state. For information on null field values, see Chapter 3, “The Elements of FMS.”

► BASIC COBOL FORTRAN PASCAL PL/1 C ☐ no, yes

These options allow you to select the programming languages for which the Forms Editor will generate field-ID files and field-values files. If `icss_edit_form` was invoked with any language options, or if the form was written out previously with any language options enabled, then the corresponding ☐ s language options are initially cycled to yes. You can change the cycle values as you wish.

► STRINGS ☐ yes, no

This option has the same effect as the `-fortran_strings` argument of the `icss_edit_form` command.

The Define/modify video display modes Request

The ☐ v option, Define/modify video display modes, allows you to change the video attributes for a region of a form.

When you issue the ☐ v request, the Forms Editor displays the form shown in Figure 4-8.

— Video Display Modes —

INTENSITY: same intensity	same underlining	same inversion
	same blinking	same blanking

Figure 4-8. Form Displayed by the ☐ v (Define/modify video display modes) Request

The five cycle fields in Figure 4-8 allow you to define or redefine the video display modes of a region of the form you are creating. The region can consist of a single field, a region with already defined modes, or a region that is highlighted when you issue the ☐ v request. Thus, the region can contain more than one field and can include part or all of the background text. ☐ v is normally used to alter the video display attributes of background text only. You can change the video display attributes of a field with the ☐ F, Add/modify field, request.

The term *same* in the default cycle values means to leave the display mode as it is currently set for the region.

The Files Produced by the Forms Editor

This section briefly describes the files produced by the Forms Editor. Subsequent sections in this chapter fully describe the form definition file, form field-values file, and field-IDs file.

The Forms Editor can generate six types of files. These types are listed in Table 4-7.

Table 4-7. Files Created by the Forms Editor

Editor-Generated Files	File Name	Description/Use
form definition	<i>form_name.form</i>	Saves form in a format that can be edited or printed.
form object module	<i>form_name.obj</i>	Bound with calling program to form an executable image.
field-values	<i>form_name.incl.language</i>	An include file that declares data-structure components corresponding to the form's fields.
field-IDs	<i>form_name_ids.incl.language</i>	An include file that assigns constant names to integer field IDs.
keystrokes	<i>_edit_form.terminal_name</i>	Saves keystrokes typed in the edit buffer.
field definition	<i>field_name</i>	Saves a field definition in a format the Forms Editor can read ((MENU) E stores a field definition; (MENU) R reads a field definition into a form).

The following subsections briefly describe these files. The contents of the form definition, field-values, and field-IDs files are described in detail later in this chapter.

The Form Definition File

A *form definition file* is a text file containing a form description. You can specify the path name of this file in the `icss_edit_form` command. If you do not specify a path name, the file is put in the current directory. The file is named `form_name.form`.

You can give the form definition file as input in a later invocation of the Forms Editor. You can also display or print it.

The contents of the form definition file are described later in this chapter under the heading "Contents of the Form Definition File."

Note: Do not give a form and a program that uses the form the same name; each requires a uniquely named object module.

The Form Object Module

A *form object module* is a description of a form that can be read by the binder. The form object module is written to the same directory as the form definition file and is named `form_name.obj`. To reference the form in a program, you must declare `form_name` as an external procedure and include the form object module in your search paths when you bind the program.

You can reference the external procedure `form_name` in the form specifier of a `perform screen initialization` statement. For information on the `perform screen initialization` statement, see Chapter 16, "Statements." For information on the form specifier, see Chapter 5, "Form Options."

Note: Do not give a form and a program that uses the form the same name; each requires a uniquely named object file.

The Field-Values File

A *field-values file* is a programming-language include file that contains declarations for a set of record elements that correspond to the fields in the form. If you specify `-into` on the `icss_edit_form` command line, or if you set the `(MENU) S PRODUCE INTO` field to yes, a field-values file is created for each language you specify in the command line or in the `(MENU) S` request. The name of the COBOL field-values file is `form_name.incl.cobol`.

You can use the field-values file to declare a record of field-value variables. For example, if the form name is `dept_info`, you can declare the record as follows:

```
01 dept_fields.
   copy 'dept_info.incl.cobol'.
```

You can reference this record in the `into` or `update form` option of screen statements. For information on these options, see Chapter 5, “Form Options.”

Using the field-values file to declare the record ensures consistent declarations in all programs using a particular form. The format of the field-values file is fully described later in this chapter under the heading “The Field-Values Include File.”

The Field-IDs File

The *field-IDs file* is a programming-language include file that defines mnemonic constants for a form’s integer field IDs. A field-IDs file is created for each language you specify in the `icss_edit_form` command line or in the `(MENU) s` form. The name of the COBOL field-IDs file is `form_name_ids.incl.cobol`.

The COBOL field-IDs file contains `%replace` statements that assign meaningful names to the field IDs. A *field ID* is a two-byte integer assigned to a field by the Forms Processor. Each field in a form has a field ID that is unique within that form. The field-IDs file permits you to refer to a field by a name, rather than by a number. The field-IDs file is fully described later in this chapter under the heading “The Field-IDs Include File.”

The Forms Editor Keystrokes File

A *Forms Editor keystrokes file* contains the keystrokes you typed in the edit buffer during the Forms Editor session. The keystrokes file is generated in your home directory and is named `_edit_form.terminal_name`, where *terminal_name* is the device name of the terminal from which you invoked the Forms Editor.

Note: Unlike a Word Processing Editor keystrokes file, the Forms Editor keystrokes file **cannot** be used to reconstruct an editing session. The keystrokes file does not contain information entered on forms for the `(MENU) F`, `(MENU) I`, `(MENU) S`, and `(MENU) V` requests. However, you can display the file to view the keystrokes you typed in the edit buffer.

The Field Definition Files

A *field definition file* contains the definition of a single field in a format that the Forms Editor can interpret. Field definition files are written to a library directory specified in the `-library` option of the `icss_edit_form` command. The file name is the name of the field. The `-library` option is described under the heading “The `icss_edit_form` Command” earlier in this chapter.

A field definition file has the same format as an entry in the field options part of a form definition file. This format is described later in this chapter under the heading “Contents of the Form Definition File.”

You create a field definition file by issuing the Enter field request (**MENU** E) in the Forms Editor. You can read in a field definition file — thereby inserting the defined field into a form you are editing — with the Read field request (**MENU** R). It is possible to write a field definition file in one editing session and read it in during another, creating identically defined fields in two or more forms. You can build a library of standard field definitions for use in different forms. The **MENU** E and **MENU** R Forms Editor requests are described under the heading “Menu Edit Requests” earlier in this chapter.

Contents of the Form Definition File

Figure 4-9 shows the contents of a form definition file for a simple menu form as it would appear if displayed on a terminal. (The appearance is slightly different if the file is printed.) The form defined by the file has two fields: `employee_number` and `department`.

```

                                PERSONNEL ADMINISTRATION MENU

Press the CANCEL key at any time to cancel a form and return to the menu.
To select an operation, press the corresponding function key.

                                UPDATE EMPLOYEE RECORD          FUNCT-1
                                UPDATE PERFORMANCE RECORD        FUNCT-2
                                UPDATE PAYROLL RECORD            FUNCT-3
                                LIST EMPLOYEES                   FUNCT-4

                                Employee Number:
                                Employee Number:   employ
                                                    #####

                                Department:
                                Department:        dep
                                                    ###
```

Figure 4-9. Sample Form Definition File

(Continued on next page)

Figure 4-9. (Continued)

```

                                FORM OPTIONS
form('per')
    highest_assigned_field_id(2)
    date_ids_reused('89-09-08 19:22:06 gmt')
    date_modified('89-09-08 20:01:03 gmt')
    clear_first
    maskkeys('EEEEBBBBBBBBBBBBBBBBBBBBBBBBBBBB')
    background_mode('low_intensity')
    required_field_mode('inverse')
    alterable
    new_mode_defaults
    currency_symbol('$')
    produce_into
    language('cobol')
    language('pl1')
    language('c')
    fortran_strings
    locked_global_features('old_features','maskkeys','displaytypes')
    locked_field_features('old_features','pos','len','help','displaytype'
        ,'window')
    locked_displaytype_features('visual','action','form_picture','cycle',
        'range','validate','required_displaytype','marked_displaytype')
    locked_global_dt_features('visual','action','form_picture','cycle',
        'range','validate','required_displaytype','marked_displaytype');

                                FIELD OPTIONS

field('employee_number') id(1) position(13, 51) length(6)
    visual('underline','high_intensity')
    action()
    picture('zzzzzz')
    help('Type the employee's number, if known.') shift_char_set(0)
    shift_flags (1);

field('department') id(2) position(14, 51) length(3)
    visual('underline','high_intensity')
    action()
    picture('999')
    help('Type the department number.') shift_char_set(0) shift_flags(1);

```

The form definition file is a description of a form that can be written and read by the Forms Editor.

The form definition file has three parts.

- The form picture
- The form options
- The field options

The *form picture* shows the layout of the form in a manner that the Forms Editor can interpret. All background text is shown, along with a representation of each field.

The *form options* part of the file specifies the form options given in the (MENU) s Forms Editor request and some internal attributes of the form.

The *field options* part of the file specifies, for each field, the field and display-type options specified with the (MENU) F Forms Editor request.

The Field-IDs Include File

A COBOL *field-IDs file* contains %replace statements. The name of the field-IDs file is *form_name_ids.incl.cobol*. For example, if the name of the form is *order_entry*, then the name of the field-IDs file is *order_entry_ids.incl.cobol*. You include the file in your program with the following statement:

```
copy 'form_name_ids.incl.cobol'.
```

The Forms Editor generates a COBOL field-IDs file if you specify the -cobol command-line option, or if you set the (MENU) s form option COBOL to yes.

In designing a form, you create fields, and the Forms Editor assigns consecutive integer identifiers called *field IDs* to these fields. The Forms Editor also assigns a unique field ID to **each** element of an array field. (The field ID of the first component in an array field is the same as the field ID of the array field.) To modify a predefined form field in a screen statement, refer to the field by its field ID. Therefore, you need to know the field ID that the Forms Editor assigns to each field and array field element.

The Forms Editor constructs a *field-ID name*, *field_name_id*, from the field name. If you specify a prefix with either the -prefix option of *icss_edit_form* or the (MENU) s PREFIX form option, the field-ID name is *prefix_field_name_id*.

The field-IDs file contains %replace statements of the following form:

```
%replace field_name_id by field_id
```


When writing a program, if you include the field-IDs file and later refer to a field by its field-ID name, the compiler replaces the field-ID name with the field ID before compiling the program. This lets you refer to a field by a convenient name (*field_name_id*) instead of a number.

For example, assume you are editing a form named *form1*, and you insert a field named *customer*. Assume also that you specified the default prefix *form_name*. The Forms Editor assigns a field ID to the field *customer* and gives this ID the name *form1_customer_id* in the field-IDs file. To reference the *customer* field in your program, use the field-ID name *form1_customer_id*.

The field-IDs file contains one `%replace` statement for each simple field (input, output, or output-only), and two for each array or window field. Of the two `%replace` statements for an array or window field, the first assigns a field-ID name to the field ID (just as for a simple field), and the second specifies either the number of elements in the array or the number of rows in the window. The field-IDs file also contains a `%replace` statement that assigns a name to the greatest field ID used by the form.

Figure 4-10 shows the general format of a VOS COBOL field-IDs file when a prefix is not specified for the form. Figure 4-11 shows the general format of a field-IDs file when a prefix is specified for the form.

```

      .
      .
      .
%replace field_name_id      by I
      .
      .
      .
%replace array_field_name_id  by J
%replace array_field_name_ct  by K
      .
      .
      .
%replace window_field_name_id by L
%replace window_field_name_ct by M
      .
      .
      .
%replace form_name_max_ids   by N

```

Figure 4-10. Generalized Field-IDs File

```

      .
      .
      .
%replace prefix_field_name_id      by I
      .
      .
      .
%replace prefix_array_field_name_id  by J
%replace prefix_array_field_name_ct  by K
      .
      .
      .
%replace prefix_window_field_name_id  by L
%replace prefix_window_field_name_ct  by M
      .
      .
      .
%replace form_name_max_ids          by N

```

Figure 4-11. Generalized Field-IDs File with a Prefix Specified

In Figures 4-10 and 4-11, *I* is the identifier of a field, *J* is the identifier of an array field, *K* is the number of elements in that array field, *L* is the identifier of a window field, and *M* is the number of rows in that window field. The value *N* is the total number of predefined fields in the form.

The following are the common uses of the declarations in a field-IDs file.

- When you use field definitions to modify a predefined form, each field definition must refer to the field by its field ID.
- Several form options take or return field IDs as arguments:
 - `getcursor (field_id_variable)`
 - `nextcursor (field_id_variable)`
 - `origin ([form_id,] window_id)`
 - `putcursor (field_id)`

- When you use the `datastates` and `displaytypes` form options, you can declare the operands for these options as follows:

```
01 data_states.
02 data_state_values          comp-4 occurs form_name_max_ids.

01 display_types.
02 display_type_values        comp-4 occurs form_name_max_ids.
```

- To alter the data state of a field, or to assign a new display type to a field, you must know the field's ID number to use it as an index into the appropriate table: `data_states` or `display_types`.

Note: If fields are deleted from an existing (previously written out) form, the field IDs assigned to those fields are not reassigned to existing fields. To reassign field IDs, thereby eliminating any gaps in the sequence of assigned IDs, delete the line `highest_assigned_field_id(n)` from the form definition file, and write out the form with the command `icss_edit_form form_name -no_edit -force_write`. Renaming the form with the following command also causes field IDs to be reassigned.

```
icss_edit_form form_name new_name -no_edit -force_write
```

The Field-Values Include File

A *field-values* file contains declarations for a set of field-value variables; include the declarations in your program with a `copy` statement.

The Forms Editor generates a VOS COBOL field-values file named `form_name.incl.cobol`, if you use the `icss_edit_form` options `-into` and `-cobol`, or if you set the **(MENU)** S form options `PRODUCE INTO` and `COBOL` to yes.

A VOS COBOL field-values file defines the items of a record, with which you can declare a record variable in your program. Every field in the form has a corresponding record member, unless you explicitly exclude a field by cycling the `in field-values` option of the **(MENU)** F form to no. If you declare a variable, *field_values_structure*, in your program, and reference that variable in an `into` or `update` form option within a `perform` screen input statement, then the Forms Processor automatically loads all form field values into *field_values_structure* when the form is submitted.

The following sequence of statements causes all form field values to be loaded into *field_values_structure* when the user submits the form.

```
01 field_values_structure.  
  copy 'form_name.incl.cobol'.      /* The field-values file */  
  .  
  .  
  .  
  perform screen input 'form_name' update (field_values_structure).
```

The record members in the field-values file can have any of the simple data types allowed by the Forms Editor, or they can be one-dimensional tables of any of these types. The following are the possible VOS COBOL data types for a simple form field.

- display pic x(N)
- display-2 pic x(N)
- comp-6 pic *numeric_picture*
- comp-5
- comp-4
- comp-3 pic *numeric_picture*
- comp-2
- comp-1

A field-values file declaration that corresponds to a simple form field has the following form:

```
20 field_name field_data_type.
```

The *simple_field_name* is the name of the form field, and *field_data_type* is the data type assigned to the field in the Forms Editor with the **(MENU)** F request.

A field-values file declaration that corresponds to an array field has the following form:

```
20 array_field_name,  
  21 array_field_name_1 field_data_type occurs number_elements.
```

The *array_field_name* is the name of the array field, and *field_data_type* is the data type assigned to the field in the Forms Editor with the **(MENU)** F request.

The following fragment uses a form named form1 and a field-values file named form1.incl.cobol.

```
01 fields.  
  copy 'form1.incl.cobol'.  
  
  perform screen initialization 'form1' into (fields) ...
```

When the perform screen initialization statement is executed, the Forms Processor loads the initial field values for form1 into the record fields.

Chapter 5:

Form Options

Form options describe characteristics that apply to an entire form, rather than to an individual field. Form options can be specified in accept or screen statements, and most can be initialized with the Set/modify form option (**MENU** S) request of the Forms Editor.

Three of the form options, *form_name*, *into*, and *update*, comprise the *form specifier*. These options specify a predefined form and a set of field-value variables for that form. The form specifier has the following syntax.

```
'form_name' [ into (field_values_structure)
              update (field_values_structure) ]
```

The form specifier must precede any other statement options.

The keyword *with* introduces the form options within an accept or screen statement. This keyword comes **after** the form specifier (if any) and before any other form options. Therefore the form options clause of an accept or screen statement has the following syntax:

```
[ 'form_name' [ into (field_values_structure)
               update (field_values_structure) ] ]
[ with form_option... ]
```

This chapter describes the form options as they appear in accept and screen statements. For information on initializing form options with the Forms Editor, see Chapter 4, "The Forms Editor."

Form Option Summary

Table 5-1 lists the form options with their arguments.

Table 5-1. The screen and accept Statement Form Options

Option	Argument	Option	Argument
beep	(<i>beep_switch</i>)	message	(<i>message_string</i>)
clear		modes	(<i>modes_array</i>)
datastates	(<i>data_states_array</i>)	name	(<i>form_name</i>)
displaytypes	(<i>display_types_array</i>)	nextcursor	(<i>field_id_variable</i> [, <i>character_position</i>])
' <i>form_name</i> '		options	(<i>option_switches</i> [, <i>option_switches</i>] ...)
formid	(<i>form_id</i>)	origin	([<i>form_id</i> ,] <i>window_id</i>)
functionkey	(<i>function_code_variable</i>)	portid	(<i>port_id</i>)
getcursor	(<i>field_id_variable</i> [, <i>character_position</i>])	putcursor	(<i>field_id</i> [, <i>character_position</i>])
into	(<i>field_values_structure</i>)	redisplay	(<i>redisplay_switch</i>)
keyused	(<i>key_code_variable</i>)	status	(<i>status_code_variable</i>)
maskkeys	(<i>mask_string</i>)	timeout	(<i>time_period</i>)
max_displaytype_id	(<i>max_display_id</i>)	update	(<i>field_values_structure</i>)
max_field_id	(<i>max_field_id</i>)		

Form Option Reference Guide

This section describes each of the form options. The options are given in alphabetical order. In the accept or screen statements, you can specify the form options in any order.

► beep (*beep_switch*)

The beep option determines whether the terminal bell sounds when the form is displayed.

The operand *beep_switch* must be a comp-4 value or the constant on or off. If *beep_switch* is 1 or on, the terminal bell sounds when the form is displayed or redisplayed. If *beep_switch* is 0 or off, the terminal bell does not sound.

If the value of *beep_switch* is 1 or on and the port is in forms input mode, the Forms Processor discards any pending input. You can use this feature to prevent the Forms Processor from applying type-ahead characters to a form. For information on forms input mode, see Chapter 14, "Subroutines."

You can specify the beep option in the following statements:

```
accept
perform screen initialization
perform screen input
perform screen output
perform screen update
```

► **clear**

Note: The clear option is obsolete and is supported in the accept statement only for compatibility with previous releases.

The clear option indicates that the entire screen or a window of the screen is to be cleared.

If you specify only the clear option in an accept statement, that statement clears the entire screen. If you specify the origin option with clear, the statement clears the specified window.

The clear option is meaningful only for the initial display of a form; it is ignored when the form is redisplayed. See the redisplay option for more information on initial display and redisplay.

You can specify that the screen or window is to be cleared before the form display within the **(MENU)**s form in the Forms Editor. For more information on the Forms Editor, see Chapter 4, “The Forms Editor.”

► **datastates (data_states_array)**

The datastates option returns the data-state switches for all the fields in a form.

The operand *data_states_array* must be a reference to a table of comp-4 values. The table must contain one element for each field in the form.

The integer in the k th element of the table specifies the data-state switches of the field with field ID k . The following table shows how the switches are encoded.

Bit	Switch Name
1	DISAPPEARING_DEFAULT
2	FIELD_VALUE_GIVEN
4	FIELD_HAS_CHANGED
16	REQUIRED_FIELD
32	INPUT_FIELD
64	NEW_DATA_IN_FIELD
128	DISABLE_ENTIRE_FIELD
256	FILTER_FOR_CONVERSION

All unused bits are reserved for future use and must be set to zero.

For an explanation of each data-state switch, see Chapter 8, “Data States.”

The Forms Editor creates a field-IDs include file to provide mnemonic names for the field IDs.

You can specify the `datastates` option in the following statements:

```
accept
perform screen initialization
perform screen input
perform screen inquire
perform screen output
perform screen update
```

You cannot use the `datastates` option with the `clear`, `modes`, or `redisplay` form options.

The `datastates` option is output-only when used in the `perform screen initialization` and `perform screen inquire` statements; the initial values in `data_states_array` are ignored. These values are also ignored in all other statements unless you set the `COPY_DATASTATE` switch in the options form option to true.

You can override specific input values in the `data_states_array` with the `datastate` field option described in Chapter 6, “Field Descriptions.”

► **displaytypes** (*display_types_array*)

The **displaytypes** option specifies or returns the display-type IDs for all fields in the form.

The operand *display_types_array* must be a reference to a table of comp-4 variables. The table must contain one element for each field in the form.

The integer in the k th element of the table specifies the display-type ID of the field with field ID k . For more information on display types, see Chapter 7, “Display Types.”

The Forms Editor creates a field-IDs include file to provide mnemonic names for the field IDs.

You can specify the **displaytypes** option in the following statements:

```
accept
perform screen initialization
perform screen input
perform screen inquire
perform screen output
perform screen update
```

You cannot use the **displaytypes** option with the **clear**, **modes**, or **redisplay** form options.

The **displaytypes** option specifies the display types on input and returns the current display types on output. For the **perform screen initialization** and **perform screen inquire** statements, the **displaytypes** option is output-only — the initial values in the table are ignored.

You can override specific input values in the *display_types_array* with the displaytype field option described in Chapter 6, “Field Descriptions.”

► **form** (*form_entry*)

The **form** option returns the entry value of a predefined form.

The operand *form_entry* must be a reference to an entry variable.

In the **perform screen inquire** statement, the **form** option returns the entry value used to invoke a predefined form.

You can specify the **form** option only in the **perform screen inquire** statement.

► 'form_name'

The 'form_name' option specifies a predefined form to be displayed and processed.

The value *form_name* must be the entry name of a predefined form.

You can specify the 'form_name' option in the following statements:

```
accept
perform screen delete
perform screen discard
perform screen initialization
perform screen input
perform screen inquire
perform screen output
perform screen save
perform screen update
```

► formid (*form_id*)

The formid option specifies the integer ID of a form.

The operand *form_id* must be an expression convertible to comp-4. In the accept and perform screen initialization statements, *form_id* must be a reference to a comp-4 variable.

In the perform screen initialization statement and in the initial-display accept statement, the formid option is output-only. In these cases, it returns a unique identifier for the form. In all other cases, the formid option is input-only and specifies which form the statement operates on.

You can specify the formid option in the following statements:

```
accept
perform screen delete
perform screen discard
perform screen initialization
perform screen input
perform screen inquire
perform screen output
perform screen save
perform screen update
```

► **functionkey** (*function_code_variable*)

The **functionkey** option returns the code associated with the function key used to submit or cancel the form.

The operand *function_code_variable* must be a reference to a comp-4 variable. The **functionkey** option is output-only. It returns the generic input request code for the key that caused return from the form. The returned value is in the range 0 through 255.

For information on the generic input requests, see the *VOS Communications Software: Asynchronous Communications (R025)*.

You can specify the **functionkey** option in the accept or perform screen input statement.

► **getcursor** (*field_id_variable* [, *character_position*])

The **getcursor** option returns the position of the cursor when the form is submitted.

The operands *field_id_variable* and, if given, *character_position* must be references to comp-4 variables. On return from the form to the application, these variables contain the final position of the cursor. The field ID of the field containing the cursor is returned in *field_id_variable*. The operand *character_position*, if specified, is set to the character position of the cursor within the field: 1 for the first character position, and so forth.

If control returns to the program as the result of a trap, the field ID returned in the **getcursor** option is the ID of the trap field.

The Forms Editor creates a field-IDs include file to provide mnemonic names for the field IDs.

You can specify the **getcursor** option in the following statements:

```
accept
perform screen initialization
perform screen input
```

► **into** (*field_values_structure*)

The **into** option returns the value specified for each field of a form.

The operand *field_values_structure* must be a reference to a record to receive the values from the fields in the form. You can use the field-values file produced by the Forms Editor to declare this record.

The **into** option is output-only; the initial values in the referenced record are ignored.

You can use the **into** option in the **accept** and **perform** screen initialization statements only. You can specify the **into** option only within the form specifier. The form specifier must include the name of a predefined form. You cannot use the **into** option with the **update** option.

When used in the **perform** screen initialization statement, the **into** option returns any initial field values specified in the Forms Editor. If no initial value is specified for a field, the null value for the field is returned. For information on null field values, see Chapter 3, "The Elements of FMS."

In the **accept** statement, the **into** option is similar to **update** with the **NO_COPY_UPDATE** options switch set to true. For further information, see the description of the **accept** statement in Chapter 16, "Statements."

► **keyused** (*key_code_variable*)

The **keyused** option returns a code associated with the key used to submit or cancel the form.

The operand *key_code_variable* must be a reference to a **comp-4** variable.

The **keyused** option is output-only. The value returned is a code identifying either the key used to submit or cancel the form, or another action that caused control to return from the form to the application. Table 5-2 lists the codes that can be returned in the **keyused** option.

Table 5-2. Codes Returned by the keyed Form Option

Code	Meaning
-9	trap on field entry
-6	form knocked down [†]
-5	form is output-only [‡]
-2	trap on field exit or vertical scroll trap
-1	form canceled or timeout occurred
0	form submitted with ENTER key
1	form submitted with function key 1
2	form submitted with function key 2
.	.
.	.
.	.
32	form submitted with function key 32

[†] See Appendix F, "Global Control Operations."

[‡] This value is returned only by the accept statement. The perform screen input statement returns -1 in the keyed option and `e$form_needs_input_field (3918)` in the status form option.

For information on enabling function keys 1 through 32, see the description of the `maskkeys` form option.

You can use the `keyed` option in the accept or perform screen input statement.

► **maskkeys** (*mask_string*)

The `maskkeys` option specifies how the Forms Processor should interpret each of the 32 generic function key requests.

The operand *mask_string* must be convertible to `pic x(32) display-2`. In the perform screen inquire statement, *mask_string* must be a reference to a variable of the type `pic x(32) display-2` or `pic x(32) display`.

Each character in the string corresponds to one of the generic function keys (function-key-1 through function-key-32). The corresponding *mask_string* character indicates what action is associated with the generic function key. The possible values are listed in Table 5-3.

Table 5-3. Characters Used in the `maskkeys` Form Option

Mask String Character	Action
b, B, or space	Sound bell and take no other action (invalid key).
c or C	Cancel the form.
e or E	Submit the form.

For example, if the first character in `mask_string` is e, then the sequence associated with the function-key-1 request causes the form to be submitted and field values to be loaded into program variables in the same way the `ENTER` key does. However, the application can differentiate between these two ways of submitting the form by using the `keyused` form option described earlier.

Similarly, if the second character in `mask_string` is c, then the sequence associated with the function-key-2 request behaves like the `CANCEL` key, except that it returns a different value to the `keyused` option.

If the third character in `mask_string` is b, then the sequence associated with the function-key-3 request does not cause the form to be submitted or canceled. If the user presses this key, the terminal bell sounds and the following message appears at the bottom of the screen:

Invalid function key.

The Forms Processor then continues to wait for a valid sequence.

The default for each generic function key is to sound the terminal bell and wait for another sequence.

Each generic function key can be associated with an actual key sequence via the terminal-type table. If you do not know which generic function keys are enabled for your terminal, or if you do not know the associated key sequences, ask your system administrator. For information on defining terminal types, see the *VOS Communications Software: Defining a Terminal Type (R096)*.

Note that if you supply a null string for the `maskkeys` option, all generic function keys are disabled. In this case, if the form does not have any input fields, it is an output-only form.

You can specify the maskkeys option in the following statements:

```
accept
perform screen initialization
perform screen input
perform screen inquire
perform screen output
perform screen update
```

► **max_displaytype_id** (*max_display_id*)

The **max_displaytype_id** option returns the greatest display-type ID value currently allocated for the form.

The operand *max_display_id* must be a reference to a comp-4 variable. The **max_displaytype_id** option is output-only and is valid only in the perform screen inquire statement.

For information on display types and how they are allocated, see Chapter 7, “Display Types.”

► **max_field_id** (*max_field_id*)

The **max_field_id** option returns the greatest field-ID value currently allocated for the form.

The operand *max_field_id* must be a reference to a comp-4 variable. The **max_field_id** option is output-only and is valid only in the perform screen inquire statement.

► **message** (*message_string*)

The **message** option specifies a message to be displayed at the bottom of the screen when the form is displayed.

The value *message_string* must be convertible to pic x(80) display-2.

If the message is too long to fit on the message line, it is truncated.

You can specify the message option in the following statements:

```
accept
perform screen initialization
perform screen input
perform screen output
perform screen update
```

► **modes** (*modes_array*)

Note: The **modes** option is obsolete and is supported in the **accept** statement only for compatibility with previous releases. New applications should use the **displaytypes** and **datastates** options instead.

The **modes** option specifies the mode values for each field in the form.

The operand *modes_array* must be a reference to a table of **comp-4** variables. The table must contain one element for each field in the form.

The *k*th element of *modes_array* corresponds to the field with field ID *k*. (The Forms Editor creates a field-IDs include file to provide mnemonic names for the field IDs.) Each element of the modes table encodes switches for the associated field. The following table shows how the switches are encoded.

Bit	Switch Name
1	BLANKED
2	BLINKING
4	INVERSE
8	UNDERLINED
16	LOW_INTENSITY
32	HIGH_INTENSITY
64	(reserved)
128	(reserved)
256	INPUT_DISABLED
512	NO_OVERLAY
1024	AUTO_TAB
2048	IMMEDIATE_RETURN
4096	NOT_EDITABLE
8192	TRAP_ON_ENTRY
16384	(reserved)
32768	(reserved)

For an explanation of each mode, see Appendix A, “The **accept** Statement.”

On initial display, the **modes** option is output-only. Field modes are returned in *modes_array*. On redisplay, the **modes** option is both input and output. Before redisplaying the form, mode values are read from *modes_array* and applied to the fields. This allows you to change field modes between the initial display and the redisplay.

To alter predefined modes on initial display, use the **mode** field option described in Chapter 6, “Field Descriptions.”

► **name** (*form_name*)

The name form option returns the name of the form.

The operand *form_name* must be a reference to a fixed- or varying-length character string variable. If the string is less than 32 characters long, some names might be truncated. The name option is output-only and is valid only in the perform screen inquire statement.

► **nextcursor** (*field_id_variable* [, *char_position_variable*])

The nextcursor option returns the logical cursor position for the next display of the form.

The operands *field_id_variable* and *char_position_variable* must be references to comp-4 variables. Both operands are output-only.

When control returns to the application from the form, the nextcursor option returns the location where the cursor would be if control had not returned. This is often the best place to put the cursor the next time the form is displayed.

In the perform screen input statement, the nextcursor option usually returns the same values as getcursor. Exceptions to this occur when control returns from the form because of a trap on field exit or a vertical scroll trap. In these cases, nextcursor returns the location the cursor would have moved to if the trap had not occurred. In the perform screen initialization statement, the nextcursor option returns an initial position for the cursor. In the perform screen inquire statement, it returns the current cursor position.

You can specify the nextcursor option in the following statements:

```
perform screen initialization
perform screen input
perform screen inquire
```

► **options** (*options_switches* [, *options_switches*] ...)

The options form option specifies a series of switches related to the form.

Each *options_switches* must be a value convertible to comp-5. In the perform screen inquire statement, *options_switches* must be a comp-5 variable. The operand is output-only in the perform screen inquire statement and is input-only in all other contexts.

Each *options_switches* encodes a set of switches for the form. The following table shows how the switches are encoded.

Bit	Switch Name
2	VERTICAL_SCROLL_TRAP
4	WIDE_CURSOR
64	VALIDATE_ERRORS_OFF
128	NO_COPY_UPDATE
256	COPY_DATASTATE
8192	CHECK_3270_FORMS_MODEL
32768	VALIDATE_ONE_FIELD
2^{28}	SPECIAL_OPTION_28
2^{29}	SPECIAL_OPTION_29
2^{30}	SPECIAL_OPTION_30

VERTICAL_SCROLL_TRAP

If this switch is true, control returns from the form to the application when the user tries to move beyond the last field or before the first field, or if the user issues a **SCROLL** (↓) or **SCROLL** (↑) request.

WIDE_CURSOR

If this switch is true, the user cannot modify input fields. When the cursor is moved into a field, the entire field is highlighted.

VALIDATE_ERRORS_OFF

If this switch is true and a trap on field exit or trap on field entry occurs, the validation routines for all fields are invoked, but validation errors are returned for the trap field only. Other validation errors are ignored.

NO_COPY_UPDATE

If this switch is true, the update form option and update field options are output-only.

COPY_DATASTATE

If this switch is true, the datastates form option is input-output, rather than output-only.

CHECK_3270_FORMS_MODEL

If this switch is true and the form employs an option that makes it unusable on a 3270, an error is indicated. If a form option makes the form unusable, the code `e$form_invalid_3270_option (3888)` is returned. If a display-type action switch makes the form unusable, the code `e$form_invalid_3270_action (3889)` is returned.

VALIDATE_ONE_FIELD

If this switch is true and a trap on field exit or a trap on field entry occurs, only the validation routine for the trap field is invoked. Any other validation routines are not invoked.

SPECIAL_OPTION_28

SPECIAL_OPTION_29

SPECIAL_OPTION_30

Specific device drivers might assign meaning to these switches. The values of these switches are passed through the Forms Processor to the driver.

The special option bits are available for driver-specific use. All other unused bits are reserved for future use and must be set to 0.

If you supply more than one *options_switches* value, then a switch is considered to be true if it is true in **any** of the values. A switch is considered false if it is false in **all** of the values.

You can specify the options form option in the following statements:

- accept
- perform screen initialization
- perform screen input
- perform screen inquire
- perform screen output
- perform screen update

You cannot use options with the clear, modes, or redisplay form options.

Note that the options form option is input-only in the perform screen initialization statement. If you want to preserve the option switch values that were set in the Forms Editor, you must use the perform screen inquire statement to obtain those values.

► `origin ([form_id,] window_id)`

The `origin` option specifies a window field in which a form is to be displayed, or it specifies that the form is to be scrolled onto the screen.

The operand *form_id*, if given, must be the form ID of an active form. The operand *window_id* must be the field ID of a window field in that form. You can obtain field IDs for a predefined form from the field-IDs include file produced by the Forms Editor.

To display a subform within a master form, use the `origin` option to specify the window field in which the form is to be displayed. For information on windows and subforms, see Chapter 11, “Windows and Subforms.”

To initialize a scrolling form, omit the *form_id* and specify 0 as the *window_id*. When the form is displayed, the screen is not cleared. Instead, text on the screen scrolls up, and the form is displayed beneath it.

► `portid (port_id)`

The `portid` option specifies which port the statement is to perform I/O on.

The operand *port_id* must be a comp-4 VOS port ID value.

If you do not specify a port ID for a statement, the terminal port is used.

The `portid` option applies only to the statement on which it is given. The value is not stored for future reference by the Forms Processor.

Note: Some device attachments support only the terminal port.

You can specify the `portid` option in the following statements:

```
accept
perform screen delete
perform screen discard
perform screen initialization
perform screen input
perform screen inquire
perform screen output
perform screen save
perform screen update
```

► `putcursor (field_id [,character_position])`

The `putcursor` option specifies where to initially position the cursor when the form is displayed.

The operands `field_id` and, if given, `character_position` must be convertible to `comp-4`.

The `field_id` specifies the ID of the field in which to put the cursor. If you supply the operand `character_position`, that value specifies the character on which to put the cursor: 1 for the first character position, and so forth.

Note: Some device drivers might ignore the value of `character_position`.

The Forms Editor creates a field-IDs include file to provide mnemonic names for the field IDs.

You can specify the `putcursor` option in the following statements:

```
accept
perform screen initialization
perform screen input
perform screen output
perform screen update
```

Note: The `nextcursor` form option returns appropriate values to be used in the `putcursor` option for the next display of a form.

► `redisplay (redisplay_switch)`

Note: The `redisplay` option is obsolete and is supported in the `accept` statement only for compatibility with previous releases.

The `redisplay` option specifies whether the form is to be redisplayed or displayed for the first time.

The operand `redisplay_switch` must be a `comp-4` value. If `redisplay_switch` is 1, a redisplay of the form is performed, rather than an initial display. The value of `redisplay_switch` must be false (0) for the first display of the form.

For information on initial displays and redispays, see Appendix A, "The `accept` Statement."

The redisplay option is valid for the accept statement only. It must **not** be used with any of the following form options:

- datastates
- displaytypes
- formid
- options
- portid
- timeout

The Forms Processor automatically detects initial display and redisplay of forms in applications that use the screen statements.

► status (*status_code_variable*)

The status option returns a VOS status code.

The operand *status_variable* must be a reference to a comp-4 variable. The status option is output-only.

If no error or exceptional condition occurs, the returned status code is 0.

For further information on handling error codes, see Chapter 10, "Error Handling and Field Validation."

You can specify the status option in the following statements:

- accept
- perform screen delete
- perform screen discard
- perform screen initialization
- perform screen input
- perform screen inquire
- perform screen output
- perform screen save
- perform screen update

► timeout (*time_period*)

The timeout option specifies the maximum amount of time the Forms Processor will wait for the user to either submit or cancel an input form.

The operand *time_period* must be convertible to comp-5.

The input timeout period is expressed in 1/1024 of a second. Therefore, the option timeout(10240) would establish a timeout period of ten seconds.

Although the timeout option affects only the accept and perform screen input statements, you can specify it in any of the following statements:

```
accept
perform screen initialization
perform screen input
perform screen inquire
perform screen output
perform screen update
```

The timeout option is input-only, except in the perform screen inquire statement, where it is output-only. In the perform screen inquire statement, the timeout option returns the timeout value currently established for the form.

► update (*field_values_structure*)

The update option specifies and returns the value of each field in the form.

The operand *field_values_structure* must be a reference to a record containing field-value variables for the form. When the form is submitted, field values are loaded into the field value variables.

The update option is usually input as well as output: values are read from *field_values_structure* and applied to the display list before the form is displayed. However, if the NO_COPY_UPDATE switch in the options form option is true, the update option is output-only.

You can specify the update option in the form specifier within the following statements:

```
accept
perform screen initialization
perform screen input
perform screen output
perform screen update
```


Chapter 6:

Field Descriptions

This chapter describes the field description clause of the accept and screen statements.

The general syntax of a field description is as follows:

```
field (field_id) [field_option] ...
```

Note: The keyword `using` must precede the first field description in an `accept` or `screen` statement.

The value *field_id* must be convertible to `comp-4` and must specify the field ID of the field. You can obtain field IDs for predefined fields from the `field-IDs` file produced by the Forms Editor (see Chapter 4, "The Forms Editor"). You can use built-in functions to allocate new field IDs, find predefined field IDs, or scan the list of allocated field IDs. For more information on built-in functions, see Chapter 15, "Built-In Functions."

The field options are described later in this chapter.

Depending on the context in which it is used, a field description performs one of the following functions:

- defines a field
- alters an existing field
- returns field information
- deletes a field.

Note: Special rules apply when using field descriptions in an `accept` statement to modify existing fields. For more information, see Appendix A, "The `accept` Statement."

Defining a Field

If you do not reference a predefined form by using the form specifier in the perform screen initialization statement, the form is created dynamically. This means that field descriptions in the perform screen initialization statement define the initial fields in the form.

Whether the form is predefined or not, the perform screen update, perform screen input, and perform screen output statements can add fields to a form after it has been initialized.

The position and length field options are required when you dynamically create a field.

Defining a Form Dynamically

If you do not reference a predefined form in a perform screen initialization statement, field descriptions within that statement define the fields of the form.

Because you cannot specify background text as such in the perform screen initialization statement, you must define each segment of background text as an output-only field and initialize it to the appropriate value. To define an output-only field, omit the update field option from the field description.

To establish an updatable input or output field, include the update field option in the field description.

Each field description must also include the position and length field options.

In the following example, a perform screen initialization statement defines a form.

```
01 fields.
    02 name           pic x(32) display-2.
    02 street         pic x(32) display-2.
    02 city           pic x(32) display-2.
    02 phone          pic s9(10) comp-6.

01 display_types.
    02 dis_type_value comp-4 occurs 8 times.

01 error_code        comp-4.
01 form_id            comp-4.
    .
    .
    .
```

(Continued on next page)

(Continued)

```

perform screen initialization with formid (form_id)
    displaytypes (display_types) status (error_code),
    using field (1) position (1, 2) length (5)
        initial ('Name:'),
    field (2) position (1, 13) length (32)
        update (name of fields),
    field (3) position (2, 2) length (7)
        initial ('Street:'),
    field (4) position (2, 13) length (32)
        update (street of fields),
    field (5) position (3, 2) length (5)
        initial ('City:'),
    field (6) position (3, 13) length (32)
        update (city of fields),
    field (7) position (4, 2) length (6)
        initial ('Phone:'),
    field (8) position (4, 13) length (10)
        update (phone of fields).

```

The form defined by this statement has four updatable fields (fields 2, 4, 6, and 8), each of which is preceded by a background label (fields 1, 3, 5, and 7). Note that field-value variables are defined only for the updatable fields.

You cannot specify the initial display types of fields you define in the perform screen initialization statement. The Forms Processor assigns one of the reserved global display types to each field as follows:

- If the field is output-only (no update field option is given), the Forms Processor assigns display-type ID 1 to the field.
- If the field has a numeric field-value variable, the Forms Processor assigns display-type ID 2 to the field.
- The Forms Processor assigns display-type ID 3 to all other fields.

If you include a `displaytypes` form option in the perform screen initialization statement, the Forms Processor returns the data-state ID for each field. If the display-types table contains extra elements, these elements are set to the null display-type ID (-32768). You can subsequently assign each field a different display-type ID if you wish. For information on display types, see Chapter 7, “Display Types.”

You cannot specify initial data states for fields defined in a perform screen initialization statement. If you include the `datastates` option in the perform screen initialization statement, the Forms Processor returns data-state values with all switches false for each element. However, if you subsequently display the form with the perform screen input statement, without the `COPY_DATASTATE` options form option switch true, that statement sets the `INPUT_FIELD` switch to true for every field that

uses the `update field` option. If you want to set other data-state switches, such as `REQUIRED_FIELD`, you must manually set the data state of each field. For information on setting data states, see Chapter 8, “Data States.”

You can display the form with a `perform screen input` statement such as the following:

```
perform screen input with formid (form_id) status (error_code),  
    using field (2) update (name of fields),  
        field (4) update (street of fields),  
        field (6) update (city of fields),  
        field (8) update (phone of fields).
```

Note that field descriptions for fields 1, 3, 5, and 7 do not appear in the `perform screen input` statement. Once a background field has been established, you need not reference it again.

If a form is defined within the `perform screen initialization` statement, you cannot use the `update form` option when you display the form. Instead, you must use the `update field` option for each updatable field.

Adding a Field Dynamically

You can dynamically add a field to a form after it is initialized. This holds true whether the form is predefined or defined within the `perform screen initialization` statement.

To add a field, you must do the following:

1. Locate an empty area of the form in which the field can be added.
2. Allocate a field ID for the field.
3. Include a field description for the field in a `perform screen update`, `perform screen input`, or `perform screen output` statement.

Typically, you must also establish a data state and a display type for the new field.

The `alloc_screen_field` built-in function allocates and returns a new field ID. For information on FMS built-in functions, see Chapter 15, “Built-In Functions.”

For example, the following fragment dynamically adds two fields to the form defined previously in this section.

```

01 ext_data_state      comp-4.
01 ext_display_type    comp-4.
01 extension_id        comp-4.
01 extension_label_id  comp-4.
01 extension_number    comp-4.
.
.
.

move !alloc_screen_field (form_id) to extension_label_id.
move !alloc_screen_field (form_id) to extension_id.

move INPUT_FIELD to ext_data_state.
move 2 to ext_display_type.

perform screen input with formid (form_id)
  options (COPY_DATASTATE) status (error_code),
  using field (2) update (name of fields)
    datastate (data_states (2))
    displaytype (display_types (2)),
  field (4) update (street of fields)
    datastate (data_states (4))
    displaytype (display_types (4)),
  field (6) update (city of fields)
    datastate (data_states (6))
    displaytype (display_types (6)),
  field (8) update (phone of fields)
    datastate (data_states (8))
    displaytype (display_types (8)),
  field (extension_label_id) position (5, 2)
    length (10) initial ('Extension:'),
  field (extension_id) position (5, 13) length (4)
    update (extension_number)
    data_states (ext_data_state)
    displaytype (ext_display_type).

```

The two fields added in this example establish an updatable field for an extension number and a background label for that field. The field IDs are obtained by invoking the `alloc_screen_field` function. Each invocation of this function returns a new field ID.

Note that, in this example, the data state for each updatable field is specified by a `datastate` field option. If you instead use a `datastates form` option to specify a data-states table, elements from that table are applied only to the previously defined fields; the data-state switches for the new fields are all set to false.

Modifying a Field

You can use a field description to modify a field in a perform screen update, perform screen input, or perform screen output statement. You can specify field options that change the following characteristics of a field:

- the field's data state (datastate field option)
- the field's display type (displaytype field option)
- the field's output value (initial or update field option)
- the field's help text (help field option).

The following example changes the help text for a field.

```
perform screen update with formid (form_id) status (error_code),  
    using field (2) help ('Enter the name of the next employee.').
```

Obtaining Field Information

You can use a field description in the perform screen inquire statement to obtain the following information about a field.

- Name (name field option)
- Position (position field option)
- Length (length field option)
- Current value (input field option)
- Data state (datastate field option)
- Display-type ID (displaytype field option)
- Help message (help field option)
- Window size (window field option; for window fields only)

The following example returns information about a field.

```
01  column          comp-4.  
01  current_value   pic x(32) display-2.  
01  field_length     comp-4.  
01  field_name       pic x(32) display-2.  
01  row              comp-4.
```

```
perform screen inquire with formid (form_id)  
    status (error_code),  
    with field (2) name (field_name)  
        position (row, column)  
        length (field_length)  
        input (current_name).
```

Deleting a Field

You can use a field description in the perform screen delete statement to remove a field from the display list. In this case, no field options are used; the syntax of the field description is simply `field (field_id)`.

The following example deletes two fields from a form.

```
perform screen delete with formid (form_id) status (error_code),  
    field (5),  
    field (6).
```

If you delete a field, that field is absent in any subsequent display of the form and the field ID is available for reuse.

Field Option Summary

Table 6-1 lists the field options. Note that not all field options are valid in all situations.

Table 6-1. The accept and screen Statement Field Options

Option	Operands
array	(<i>number_rows</i> , <i>number_columns</i>)
center	
cycle	(<i>cycle_value</i> [, <i>cycle_value</i>] ...)
datastate	(<i>data_state_switches</i>)
displaytype	(<i>display_type_id</i>)
given	(<i>values_count</i>)
help	(<i>help_message</i>)
initial	(<i>initial_value</i>)
input	(<i>field_value_variable</i>)
left	
length	(<i>field_length</i>)
mode	(<i>mode_switches</i>)
name	(<i>field_name</i>)
picture	(<i>field_picture</i>)
position	(<i>line</i> , <i>column</i>)
range	(<i>low_bound</i> , <i>high_bound</i>)
redisplayfield	(<i>redisplay_field_switch</i>)
required	
right	
shift	(<i>character_set_id</i>)
unshift	(<i>character_set_id</i>)
update	(<i>field_value_variable</i>)
validate	(<i>validation_entry</i>)
window	(<i>number_lines</i> , <i>number_columns</i>)

Field Option Reference Guide

This section describes each field option and explains how each can be used. The field options are listed in alphabetical order.

► array (*number_rows*, *number_columns*)

The array option groups a set of similar fields into an array field.

The operands *number_rows* and *number_columns* must be positive integer values convertible to comp-4.

The array field option has two uses: to define a set of similar fields, and to group a set of previously defined fields.

If you use the array option in a field description that defines a field, that field description defines a set of similar fields. In this case, the position field option indicates the position of the topmost, leftmost of the individual fields. The other fields are laid out as indicated by *number_rows* and *number_columns*. There is always one space between rows and between columns.

After an array field is defined, the individual elements of the array are treated the same as individually defined fields. Each element has a unique field ID and can be modified without changing the other elements of the array.

You can also use the array option to group a set of individual fields that are already defined. If you specify the array option for a field that is already defined, that field and fields with sequentially following field IDs are treated as an array field. The number of fields included in the array is the product of *number_rows* and *number_columns*. Any field options you give with the array option are applied to all fields in the array. If the field description includes the update field option, the variable referenced in that option must be a table.

In the following example, six fields are grouped into an array.

```
01 array_var.
  02 array_values      pic x(12) display occurs 6 times.
      .
      .
      .
      perform screen input with status (error_code),
         using field (5) array (6, 1) displaytype (11)
         update (array_var).
```

The perform screen input statement in the example establishes a common display type for the six fields with IDs 5 through 10. The field values for these fields are returned in the table array_var.

Grouping existing fields into an array field does not change the location of the fields.

The field-value variable for an array field can be either a one-dimensional or two-dimensional table. The number of elements in the field-value variable must be the same as the number of elements in the array field. However, the dimensions of the table do not have to be the same as *number_rows* and *number_columns*. In the preceding example, the field-value variable could have been defined as follows:

```
01 array_var.
  02 array_rows      occurs 3 times.
  03 array_values    pic x(12) display occurs 2 times.
```

Note: In the obsolete accept statement, you can use the array field option only when defining a field. The array option is then implicitly applied to any reference to that field. For information on the accept statement, see Appendix A, “The accept Statement.”

► center

Note: The center field option is obsolete and is supported in the accept statement only for compatibility with previous releases. In new applications, use the `CENTER_FIELD_DATA` or `NOTRIM_FIELD_DATA_SPACES` display-type visual switches instead. See the description of the visual option in Chapter 7, “Display Types.”

For an input field, the center option changes the handling of leading spaces in a field value. For an output-only field, the center option forces the field value to be centered in the field.

The center, left, and right field options are mutually exclusive. For numeric fields and for fields that have a picture, some device drivers might ignore all these options. (Any field that has a numeric picture or that has a field-value variable with a numeric data type is a numeric field. See Chapter 9, “Field Pictures and Filtering.”)

The center option inhibits the trimming of leading spaces from an input field value. Ordinarily, leading spaces are trimmed from the left of an input field value before the value is validated and loaded into a program variable. Spaces are also trimmed from the left of an output value before it is displayed in the field. If the center option is specified, this trimming does not occur.

Note: Specifying the center option does not center the initial output value of an input field.

► `cycle (cycle_value [,cycle_value] ...)`

Note: The `cycle` field option is obsolete and is supported in the `accept` statement only for compatibility with previous releases. In new applications, use the `cycle display-type` option instead. See Chapter 7, “Display Types.”

The `cycle` field option specifies that a field is a cycle field.

Each `cycle_value` must be a valid value for the field.

The `cycle` option establishes a set of allowable field values. This set of values is called the *cycle list*. A field with a cycle list is called a *cycle field*. On input, the user can choose one of the cycle values by using the `(CYCLE)`, `(CYCLE BACK)`, or left and right arrow keys.

Commonly, the first value in the cycle list appears in the field on initial display. See the discussions of initial output values in Chapter 3, “The Elements of FMS,” and in Appendix A, “The `accept` Statement.”

You can use the `cycle` option when defining a form dynamically. You can also use the `cycle` option to modify an existing field if one of the following is true.

- The field was not defined as a cycle or range field.
- The field was defined as a cycle field, and the form was defined as alterable by `accept`.

For information on predefined forms and the `ALTERABLE BY ACCEPT` option, see Chapter 4, “The Forms Editor.”

► `datastate (data_state_switches)`

The `datastate` field option specifies the settings of the data-state switches for a field.

The operand `data_state_switches` must be a reference to a `comp-4` variable.

The data-state switches for the field are encoded in the bits of `data_state_switches`. The following table shows how the switches are encoded.

Bit	Switch Name
1	DISAPPEARING_DEFAULT
2	FIELD_VALUE_GIVEN [†]
4	FIELD_HAS_CHANGED
16	REQUIRED_FIELD
32	INPUT_FIELD
64	NEW_DATA_IN_FIELD
128	DISABLE_ENTIRE_FIELD
256	FILTER_FOR_CONVERSION [†]

[†] The FIELD_VALUE_GIVEN and FILTER_FOR_CONVERSION switches are output-only.

Constants for these switch values are defined in the include file
(master_disk)>system>include_library>form_datastate.incl.cobol.

You can use the datastate option to obtain the data-state switches for a field. When altering a field, you can also use the datastate option to set some of the data-state switches. To change data-state values, you must set the COPY_DATASTATE options form option switch to true.

For information on data states, see Chapter 8, “Data States.”

You can use the datastate field option with the array field option to set the data states for each element of the array. However, the value returned in the datastate field option for an array field is unpredictable. To obtain data-state values for array elements, use the datastates form option described in Chapter 5, “Form Options.”

The data-state value specified in the datastate field option overrides a data-state value specified for the field in the datastates form option.

You **cannot** use the datastate option with the modes form option or with any of the following field options:

- center
- cycle
- given
- mode
- left
- picture
- range
- redisplayfield
- required
- right
- validate

► **displaytype** (*display_type_id*)

The **displaytype** field option specifies the display type for a field.

The operand *display_type_id* must be convertible to comp-4. In a perform screen inquire statement, *display_type_id* must be a reference to a comp-4 variable. In the perform screen inquire statement, the **displaytype** option is output-only. In all other contexts, it is input-only.

In a perform screen inquire statement, the **displaytype** option returns the display-type ID for the field. In all other contexts, the **displaytype** field option specifies the display-type ID for the field.

A display type must be defined before it is referenced in a **displaytype** field option. For information on display types, see Chapter 7, "Display Types."

A display type specified in the **displaytype** field option overrides a display type specified for the field in the **displaytypes** form option.

You **cannot** use the **displaytype** option with the **modes form** option or with any of the following field options:

- center
- cycle
- given
- mode
- left
- picture
- range
- redisplayfield
- required
- right
- validate

► **given** (*values_count*)

Note: The given field option is obsolete and is supported in the accept statement only for compatibility with previous releases. In new applications, use the `FIELD_VALUE_GIVEN` switch in the datastate field option instead. See Chapter 8, “Data States.”

The given option indicates whether the user specified a value for a field.

The operand *values_count* must be a reference to a comp-4 variable. The given option is output-only.

The given option returns the value 0 if the user does not give a value for the field. If the field is a simple (non-array) field and the user supplies a value, the given option returns the value 1. If the field is an array field, the given option returns the number of array elements for which the user has given a value.

► **help** (*help_message*)

The help option specifies help text for a field.

The operand *help_message* must be a character-string value. In the perform screen inquire statement, *help_message* must be a reference to a character-string variable.

In the perform screen inquire statement, the help option returns the help message for the field. In all other contexts, the help option establishes the help message for the field.

The help message for a field is displayed on the terminal status line when the user positions the cursor to the field and presses the **HELP** key.

If the help message is too long to be displayed on the terminal status line, only the leftmost characters are displayed.

► $\left\{ \begin{array}{l} \text{initial (initial_value)} \\ \text{initial_value_string} \end{array} \right\}$

The initial field option specifies the initial output value for a field.

The operand *initial_value*, if given, must be convertible to a valid value for the field. The operand *initial_value_string*, if given, must be a character-string constant that is convertible to a valid value for the field.

If you specified an initial value for the field in the Forms Editor and the form is not alterable by accept, then the initial field option in the perform screen initialization statement is ignored. In this case, the initial value specified in the Forms Editor is used.

Note: The initial value for a cycle field must be one of the values in the cycle list.

► input (*field_value_variable*)

The input field option returns the current value of a field.

The operand *field_value_variable* must be a reference to a variable that can receive the value of the field. The input field option is output-only.

The input field option can only be used in the perform screen inquire statement.

► left

Note: The left field option is obsolete and is supported in the accept statement only for compatibility with previous releases. In new applications, use the LEFT_JUSTIFY_FIELD_DATA display-type visual switch instead. See Chapter 7, "Display Types."

The left field option specifies that a field is to be left-justified.

If the field has a field picture, for some device drivers the picture determines the justification of the field. In this case, the left option is ignored. For information on field pictures, see Chapter 9, "Field Pictures and Filtering."

The left option might also be ignored if the field is numeric. For some device drivers, numeric fields are always right-justified. (A numeric field is any field having a numeric field picture or having a field-value variable with a numeric data type and no field picture.) The default justification for alphanumeric fields is left.

On input, the first character typed by the user in a left-justified field appears in the leftmost character position. The cursor moves to the right as each character is typed so that subsequent characters fill to the right.

On output, the field value for a left-justified field is displayed in the leftmost positions of the field.

By default, the initial editing mode for a left-justified field is overlay mode. You can change the initial editing mode by setting either the `NO_OVERLAY` mode bit or the `FORCE_INSERT_MODE` display-type action switch. (See Chapter 7, “Display Types”.) The user can change the editing mode of a field with the `(INSERT/OVERLAY)` key.

► `length (field_length)`

The `length` field option specifies the length of a field.

The operand `field_length` must be convertible to `comp-4`. In the `perform screen inquire` statement, `field_length` must be a reference to a `comp-4` variable.

For a simple field, the `length` option specifies the number of visible character positions in the field. Note that a character from a double-byte character set requires two character positions. For an array field, the `length` option specifies the length of each element. In the `perform screen inquire` statement, `length` returns the length of the field or array elements.

The `length` option is required when defining a field.

You should not define a field that is too long to fit on the screen.

► `mode (mode_switches)`

Note: The `mode` field option is obsolete and is supported in the `accept` statement only for compatibility with previous releases. In new applications, use the `datastate` and `displaytype` field options instead. These options are described earlier in this chapter.

The `mode` field option specifies the mode settings for a field.

The operand `mode_switches` must be convertible to `comp-4`. The `mode` option is input-only.

The mode field option has two uses:

- to set the mode switches when defining a field
- to alter the mode switches of a predefined field on initial display.

You can use the mode option to alter the mode switches on initial display only if the form is alterable by accept. When used in this way, the mode option must be the only field option in the field description. Such a field description is read on initial display and ignored on redisplay.

The following table shows how the mode switches are encoded in the bits of *mode_switches*.

Bit	Switch Name
1	BLANKED
2	BLINKING
4	INVERSE
8	UNDERLINED
16	LOW_INTENSITY
32	HIGH_INTENSITY
64	(reserved)
128	(reserved)
256	INPUT_DISABLED
512	NO_OVERLAY
1024	AUTO_TAB
2048	IMMEDIATE_RETURN
4096	NOT_EDITABLE
8192	TRAP_ON_ENTRY
16384	(reserved)
32768	(reserved)

All unused bits are reserved and must be set to 0. For an explanation of each mode, see Appendix A, "The accept Statement."

To modify field modes on redisplay, use the modes form option.

► name (*field_name*)

The name field option returns the name of a field.

The operand *field_name* must be a reference to a character-string variable. The name option is output-only.

The name option is valid only in the perform screen inquire statement and only for fields predefined by the Forms Editor.

A field's name is the name specified for the field in the Forms Editor. This is the same name used to name the associated variable in the field-values include file.

► picture (*field_picture*)

Note: The picture field option is obsolete and is supported in the accept statement for compatibility with previous releases. In new applications, use the picture display-type option instead. See Chapter 7, "Display Types."

The picture option specifies a field picture, or template, for a field.

The operand *field_picture* must be convertible to a character string that is no longer than the field. The picture field option is input-only.

Each character in *field_picture* corresponds to a character position in the field and defines the valid characters for that field position. If *field_picture* is shorter than the field, the Forms Processor automatically extends the field picture to the length of the field.

The following table lists the valid picture characters.

Picture Character	Meaning
A or a	Allow space or hyphen.
B or b	Insert literal space.
L or l	Allow letter, digit, or space; convert letter to lowercase.
U or u	Allow letter, digit, or space; convert letter to uppercase.
X or x	Allow letter, digit, or space.
Z or z	Allow digit or hyphen (negative sign); suppress leading zeros.
9	Allow digit or hyphen (negative sign); display leading zeros.
.	Fix decimal point location. [†]
,	Fix digit-grouping character location. [†]
-	Insert literal hyphen.
/	Insert literal slant.

[†] The meaning of the period and comma can be reversed with the `decimal is comma` option in the Forms Editor (**MENU**) `s` form.

For information on field pictures, see Chapter 9, “Field Pictures and Filtering.”

► `position (line, column)`

The `position` option specifies the position of a field within the form.

The operands `line` and `column` must each be convertible to `comp-4`. In the `perform screen inquire` statement, `line` and `column` must each be a reference to a `comp-4` variable. The `position` option is input-only, except in the `perform screen inquire` statement, where it is output-only.

The position of a simple field is the line and column of the leftmost character position in the field. The position of an array or window field is the line and column of the leftmost character position of the topmost, leftmost element.

In the `perform screen inquire` statement, `position` returns the position of the field.

The `position` option is required when defining a field.

The position you specify for the field must be within the limits of the terminal screen.

► **range** (*low_bound*, *high_bound*)

Note: The range field option is obsolete and is supported in the **accept** statement only for compatibility with previous releases. In new applications, use the range display-type option instead. See Chapter 7, “Display Types.”

The range field option specifies a minimum and maximum value for a field.

The operands *low_bound* and *high_bound* must each be convertible to the data type of the field-value variable for the field. Both values are input-only.

The range option restricts field values by specifying a range of valid values. When the user submits the form, the Forms Processor checks that the input value is within the specified bounds. If the value is out of range, the Forms Processor displays an error message and positions the cursor to the field.

The Forms Processor performs the range test after checking that the value conforms to the field picture (if any) and before calling the field validation routine (if any).

When altering an existing field, you can use the range option to define or alter the bounds only if one of the following is true.

- The field was defined with neither a range restriction nor a cycle restriction.
- The field was defined with a range restriction, and the form is alterable by **accept**.

► **redisplayfield** (*redisplay_field_switch*)

Note: The **redisplayfield** field option is obsolete and is supported in the **accept** statement only for compatibility with previous releases.

The **redisplayfield** option specifies whether to display the initial output value for a field or to display the current field value.

The operand *redisplay_field_switch* must be a comp-4 value.

If *redisplay_field_switch* is true (1) on redisplay, the current value of the field-value variable is ignored on input, and the initial output value is displayed in the field. If *redisplay_field_switch* is false (0) on redisplay, the current value of the field-value variable is displayed in the field.

The **redisplayfield** option is ignored on initial display.

For information on initial display versus redisplay, see the description of the **accept** statement in Chapter 16, “Statements.” For information on initial output values, see Chapter 3, “The Elements of FMS,” and Appendix A, “The **accept** Statement.”

► required

Note: The required field option is obsolete and is supported in the accept statement only for compatibility with previous releases. In new applications, use the `REQUIRED_FIELD` data-state switch instead. See Chapter 8, "Data States."

The required field option specifies that a field is required.

If a field is required, the user cannot submit the form while the field contains its null value. If the user attempts to do so, an error message is displayed and the cursor is positioned to the field.

If you use the required field option to alter a field that was predefined by the Forms Editor, the visual attributes of the field are changed in accordance with the required field toggles specified in the Forms Editor.

For information on null field values, see Chapter 3, "The Elements of FMS." For information on the Forms Editor, see Chapter 4, "The Forms Editor."

► right

Note: The right field option is obsolete and is supported in the accept statement only for compatibility with previous releases. In new applications, use the `RIGHT_JUSTIFY_FIELD_DATA` display-type visual switch instead. See Chapter 7, "Display Types."

The right field option specifies that a field is to be right-justified.

If the field has a field picture, for some device drivers the picture determines the justification of the field. In this case, the right option might be ignored. For information on field pictures, see Chapter 9, "Field Pictures and Filtering."

For some device drivers, numeric fields are always right-justified. (A numeric field is any field that has a numeric field picture or that has a field-value variable with a numeric data type and no field picture.) The default justification for alphanumeric fields is left.

On input, the cursor appears at the right of a right-justified field. The first character typed by the user appears in the rightmost character position. As subsequent characters are typed, characters in the field move left. Thus, each new character initially appears in the rightmost position. The cursor does not move unless the user explicitly moves it with the cursor motion keys.

On output, the field value for a right-justified field is displayed in the rightmost positions of the field.

By default, the initial editing mode for a right-justified field is insert mode. You can change the initial editing mode by setting the `FORCE_OVERLAY_MODE` display-type action switch. The user can change the editing mode of a field with the `(INSERT/OVERLAY)` key.

For information on the `FORCE_OVERLAY_MODE` switch, see Chapter 7, “Display Types.”

► `shift (character_set_id)`

The `shift` field option establishes a default supplemental character set for a field and specifies how characters from supplemental character sets are represented within the field value.

The operand `character_set_id` must be a `comp-4` value.

The value of `character_set_id` indicates the default character set for the field. The following table lists the allowed values and their meanings.

Character Set ID	Character Set Name
0	none
1	LATIN_1_CHAR_SET
2	KANJI_CHAR_SET
3	KATAKANA_CHAR_SET
4	HANGUL_CHAR_SET

Constants for these values are defined in the file
(master_disk)>system>include_library>char_sets.incl.cobol.

The `shift` option specifies the default supplemental character set for the field. It also indicates that characters from any other supplemental character sets are preceded by single-shift characters on input and must be preceded by single-shift characters on output. Characters from the default character set need not be preceded by shifts on input and are not preceded by shifts on output.

If you specify `shift(0)`, you must precede all characters from supplemental character sets by single-shift characters on input, and the Forms Processor precedes all supplemental characters by single-shift characters on output.

Note: The `character_set_id` value is input-only in all contexts — including the `perform screen inquire` statement.

If you omit both the `shift` field option and the `unshift` field option, the default is `shift(1)`.

For more information on supplemental character sets, see the section “International Character Set Support” in Chapter 3, “The Elements of FMS.”

► `unshift (character_set_id)`

The `unshift` field option establishes the only supplemental character set allowed for a field and specifies how supplemental characters are represented within the field value.

The operand `character_set_id` must be a `comp-4` value.

The value of `character_set_id` indicates the character set for the field. The following table lists the allowed values and their meanings.

Character Set ID	Character Set Name
0	none
1	LATIN_1_CHAR_SET
2	KANJI_CHAR_SET
3	KATAKANA_CHAR_SET
4	HANGUL_CHAR_SET

Constants for these values are defined in the file
(master_disk)>system>include_library>char_sets.incl.cobol.

The `unshift` option specifies the only valid supplemental character set for the field and indicates that the value stored in the field-value variable is to contain no shift characters.

On input, all supplemental characters are assumed to be of the specified character set. The user cannot submit a field value containing characters from any other supplemental sets. If the user attempts to enter such a value, an error message is displayed, and the cursor is positioned to the field.

On output, the value returned to the field-value variable does not contain any shift characters. All supplemental characters in the value are of the specified character set.

If you specify `unshift(0)`, the field cannot contain any supplemental characters.

Note: The `character_set_id` value is input-only in all contexts — including the `perform screen inquire` statement.

If you omit both the `shift` field option and the `unshift` field option, the default is `shift(1)`.

For more information on supplemental character sets, see the section “International Character Set Support” in Chapter 3, “The Elements of FMS.”

► **update** (*field_value_variable*)

The update field option specifies the field-value variable for a field.

The operand *field_value_variable* must be a reference to a variable that can receive the value of the field.

If the NO_COPY_UPDATE switch in the options form option is true, the update field option is output-only. In all other cases, it is used for both input and output.

On input, the value of *field_value_variable* is converted (if necessary) to the data type of the field and displayed in the field.

On output, the value of the field is converted (if necessary) to the data type of *field_value_variable* and assigned to that variable. If the field value cannot be converted to the data type of *field_value_variable*, the form cannot be submitted; an error message is displayed, and the cursor is positioned to the field.

You can use the update field option when defining or altering an input field. If a perform screen input statement does not contain an into or update form option, you must use the update field option for each input field in the form; otherwise, the field value supplied by the user is lost.

If you omit the update option when defining a field, the field defaults to output-only.

► **validate** (*validation_entry*)

Note: The validate field option is obsolete and is supported in the accept statement only for compatibility with previous releases. In new applications, use the validate display-type option instead. See Chapter 7, “Display Types.”

The validation field option establishes a validation routine for a field.

The operand *validation_entry* must be a reference to an entry value.

The validation routine is invoked to check the field value when the user submits the form.

When the form is submitted, the Forms Processor first checks that the field value conforms to the field picture (if any). It then checks that the value is within the range specified for the field (if any). It then invokes the validation routine (if any).

For information on writing validation routines, see Chapter 10, “Error Handling and Field Validation.”

► **window** (*number_lines*, *number_columns*)

The window field option defines a window field.

The operands *number_lines* and *number_columns* must be comp-4 values. In the perform screen inquire statement, both operands must be references to comp-4 variables. Both values are output-only in the perform screen inquire statement and input-only in all other contexts.

The window field option is valid when defining a new field and in the perform screen inquire statement. When defining a window field, you must use the position field option to specify the location of the window. The length option does not apply to window fields.

For information on window fields, see Chapter 11, “Windows and Subforms.”

)

)

)

)

)

Chapter 7:

Display Types

This chapter describes field display types. It explains the different classes of display types and how display types are defined and manipulated. The display-type options are described alphabetically at the end of the chapter.

A *display type* is a combination of specific field attributes that can be referenced by an associated integer value. This integer value is the *display-type ID*.

Each field in a form has an associated display type. Often, several fields in a form share a common display type. You can change a field's attributes by assigning it a different display type. If several fields share a display type, you can alter the attributes of all those fields by modifying the shared display type.

You initially set the display-type attributes of each field in the new Forms Editor. The Add/modify field menu includes a section of display-type options. The Forms Processor assigns an ID to each unique set of display-type attributes you specify. Display types created in the Forms Editor are called *predefined display types*. If two fields have identical display-type attributes, they share a display-type ID.

Within the application program, you can use the `displaytypes` form option in the `perform` screen initialization statement to obtain the predefined display-type ID for each field. You can use the `displaytypes` form option in subsequent `perform` screen input, `perform` screen output, and `perform` screen update statements to specify different display types. For further information on the `displaytypes` form option, see Chapter 5, "Form Options."

You can use the `displaytype` field option to specify a new display type for a specific field. See the description of the `displaytype` field option in Chapter 6, "Field Descriptions."

Display-Type Descriptions

In an accept or screen statement, the display-type description clause has the following syntax:

```
displaytype (display_type_id) [display_type_option] ...
```

The operand *display_type_id* must be convertible to comp-4 and must specify the display type's ID.

Note: The keyword giving must precede the first display-type description in an accept or screen statement.

The display-type options are listed in Table 7-1.

Table 7-1. The Display-Type Options

Display-Type Option	Operands
action	([<i>action_switches</i>] [, <i>update_switches</i>] ...)
charset	(<i>character_set_id</i>)
cycle	({ <i>cycle_value</i> [, <i>cycle_value</i>] ... <i>cycle_value_array</i> [, <i>value_count</i>] })
cycle_array	(<i>cycle_value_array</i> , <i>value_count_variable</i>)
picture	(<i>picture</i>)
range	([<i>min_value</i>] [, <i>max_value</i>])
validate	(<i>validation_entry</i>)
visual	([<i>visual_switches</i>] [, <i>update_switches</i>] ...)

Each of the display-type options is described later in this chapter.

Display-type descriptions can perform the following functions:

- Define a display type. You can define a new display type in the display-type description clause of the accept, perform screen update, perform screen initialization, perform screen input, and perform screen output statements. If the display-type ID you reference in the display-type description is not in use, then the clause defines a new display type having that ID.
- Alter an existing display type. You can alter a display type in the display-type description clause of the accept, perform screen update, perform screen initialization, perform screen input, and perform screen output statements. If the display-type ID you reference is in use, then that clause

modifies the display type. Note that the modification automatically affects all fields that reference the display type.

- Obtain information about a display type. You can obtain information about a display type with a display-type description clause in the `perform screen inquire` statement. The display-type ID must reference a currently defined display type.
- Delete a display type. You can remove a display type from the display list by referencing it in a display-type description clause in a `perform screen delete` statement. The display-type ID must reference a currently defined display type that is not referenced by any existing field. The display-type description in the `perform screen delete` statement must not contain any display-type options.

Note: Some of these operations are not allowed for all display types. The classes of display types and the restrictions on each are described in the following section.

Classes of Display Types

A display type must be defined before you reference it in either the `displaytypes` or `displaytype` option. There are three classes of display types which are defined in different ways.

- Predefined display types
- Global display types
- Temporary display types

There are two subclasses of global display types: reserved and programmer-defined.

Table 7-2 summarizes the distinctions among the classes of display types.

Table 7-2. The Display-Type Classes

Class	Defined by	Scope	Alterable	Display-Type IDs
Predefined	the Forms Editor	local to form	no	negative integers
Global Reserved Programmer -defined	the Forms Processor the program	global to port global to port	no yes	0 through 10 11 through 16,383
Temporary	the Forms Processor	local to form	yes	16,384 through 32,767

Predefined Display Types

Display types established in the Forms Editor are called *predefined display types*. Predefined display types have negative IDs and are local to the form for which they are defined.

You specify the attributes of a predefined display type when you define a field with the Add/modify field option (**MENU** F) in the new Forms Editor. For information on this option, see Chapter 4, "The Forms Editor."

You cannot alter or delete predefined display types within an application. To change the display characteristics of a predefined field, assign it a different display type.

Global Display Types

Global display types are global to a port and remain valid until the port is detached. Some global display types are defined by the Forms Processor; others can be defined by the programmer. Global display types have positive display-type IDs. Display types defined by the programmer can be altered dynamically by the application.

Display-type IDs 0 through 10 are reserved for global display types defined by the Forms Processor. You cannot alter or delete these display types.

The following table lists the global display types that are currently defined.

Display-Type ID	Description
0	The display type of the error message field. [†]
1	The default display type for alphanumeric fields.
2	The default display type for numeric fields.
3	The default display type for output fields.

[†] Display-type 0 might have limited effect for some device drivers.

The display-type IDs 4 through 10 are reserved for future use.

You can create your own global display types within the application program. You must assign IDs in the range 11 to 16,383, inclusive, to these display types. You can create global display types in two ways:

- by referencing the display-type ID in a display-type description in a perform screen initialization, perform screen update, perform screen input, or perform screen output statement. (Display-type descriptions are discussed later in this chapter.)

- by obtaining the display-type ID from the `alloc_screen_displaytype` built-in function.

You should always assign display-type IDs sequentially. Using a larger display-type ID might cause the Forms Processor to waste storage space. If you know that certain display types must be defined within a program, you should define constants for those display-type IDs, starting with 11, and create the display types in perform screen update statements. If other display types need to be defined later in the application, use the `alloc_screen_displaytype` built-in function to obtain the next available display-type ID. The built-in functions are described in Chapter 15, “Built-In Functions.”

You can alter the characteristics of any global display type you define in an application program. Changing a global display type effectively changes the attributes of all fields that use that display type on that port.

Temporary Display Types

Temporary display types are created by the Forms Processor for a form created with the old Forms Editor or old-style accept statement. Temporary display types have display-type IDs in the range 16,384 to 32,767, inclusive.

Temporary display types are local to the form for which they are defined.

Setting the Action and Visual Attributes

You can specify attributes that affect the behavior and visual appearance of a field with the action and visual display-type options, respectively. Each of these options specifies settings for a series of switches encoded in a comp-5 value.

Each visual and action switch is explained later in this chapter.

To set the visual switches, you can specify a single value that gives the appropriate settings for each switch. For example, the following statement creates a display type with the blinking and underlined visual attributes.

```
perform screen update with status (status_code),
    giving displaytype (11) visual (10).
```

Rather than giving a single value for the switches, you can specify a base value and one or more sets of update switches to be applied to that base. An update switches value is always preceded by a comma. For example, the following fragment is equivalent to the previous example.

```
%replace BLINKING_VISUAL          by 2
%replace INVERSE_VISUAL           by 4
%replace LOW_INTENSITY_VISUAL     by 16
%replace UNDERLINED_VISUAL       by 8
.
.
.
```

```
perform screen update with status (status_code),
    giving displaytype (11)
    visual (BLINKING_VISUAL, UNDERLINED_VISUAL).
```

In this example, the value 2 (BLINKING_VISUAL) is specified as the base value for the visual option. The update value 8 (UNDERLINED_VISUAL) specifies an additional switch to be set to true. An update value can specify more than one switch to be set to true. For example, the update value 20 would set the INVERSE_VISUAL and LOW_INTENSITY_VISUAL switches to true. (You could achieve the same result by specifying two update values.)

When updating an existing display type, you can omit the base value. The current switch settings for the display type are then used as the base. For example, the following statement updates the display type defined in the previous example.

```
perform screen update with status (status_code),
    giving displaytype (11)
    visual ( , INVERSE_VISUAL, LOW_INTENSITY_VISUAL).
```

This example sets the INVERSE_VISUAL and LOW_INTENSITY_VISUAL switches to true. The BLINKING_VISUAL and UNDERLINED_VISUAL switches also remain true.

If an update value is preceded by a circumflex (^), it specifies switches to be set to false. The following statement updates the display type defined in the previous example.

```
perform screen update with status (status_code),
    giving displaytype (11) visual ( , ^BLINKING_VISUAL).
```

This example sets the BLINKING_VISUAL switch to false for the display type. All other switches are unchanged.

The syntax of the action attribute is similar to that of the visual attribute. For example, the following fragment defines a display type with the FORCE_INSERT_MODE

and TRAP_ON_FIELD_EXIT action switches true.

```
%replace FORCE_INSERT_MODE          by 8
%replace TRAP_ON_FIELD_EXIT        by 2
.
.
.

perform screen update with status (status_code),
    giving displaytype (12)
    action (FORCE_INSERT_MODE, TRAP_ON_FIELD_EXIT).
```

You can obtain the current settings of the action or visual switches by using the action or visual option in a display-type description of a perform screen inquire statement, as in the following example.

```
01  action_switches    comp-5.
01  visual_switches    comp-5.
.
.
.

perform screen inquire with status (status_code),
    giving displaytype (12) action (action_switches)
    visual (visual_switches).
```

In the perform screen inquire statement, the action and visual options take only one operand and are output-only.

Cycle Display Types

If you specify the cycle option for a display type, that display type is a *cycle display type*. Any field using that display type is a *cycle field*.

The cycle option establishes a set of allowable field values. This set of values is called the *cycle list*. On input, the user can choose one of the cycle values by using the **(CYCLE)**, **(CYCLE BACK)**, or left and right arrow keys (**←** and **→**).

Commonly, the first value in the cycle list appears in the field on initial display. See the discussion of initial output values in Chapter 3, "The Elements of FMS."

You can define the cycle list either in the Forms Editor or with the cycle display-type option in the application program. For information on the Forms Editor, see Chapter 4, "The Forms Editor." The cycle values can be of any data type convertible to strings.

You can also use the `cycle display-type` option to change the cycle list of a previously defined display type.

In the `cycle display-type` option, you can specify the cycle list in two ways. You can list the cycle values individually, or you can specify a table that contains the cycle values. If you specify the table, you can also specify the number of cycle values to be taken from the table.

The following example defines a simple cycle display type.

```
perform screen update with status (status_code),  
    giving displaytype (11) cycle ('add', 'change', 'remove').
```

The display type defined in the preceding example has three cycle values: `add`, `change`, and `remove`. The following example defines the same display type using the other syntax of the `cycle` option.

```
01  cycle_values.  
02  cycle_table    pic x(12) display-2 occurs 10.  
  
01  number_values.  
    .  
    .  
    .  
  
    move 'add' to cycle_values (1).  
    move 'change' to cycle_values (2).  
    move 'remove' to cycle_values (3).  
    move 3 to number_values.  
  
perform screen update with status (status_code),  
    giving displaytype (11)  
        cycle (cycle_values, number_values).
```

In this example, because `number_values` is 3, the first three elements of the table are used as the cycle list. The other elements of the table are ignored. If you omit the second operand of the `cycle` option, **all** elements of the table are taken as cycle values.

Subsequently changing the values of either `number_values` or the table elements does **not** change the cycle list for the display type. To change the cycle list, you must modify the display type in another display-type description, as in the following example.

```

move 'write' to cycle_values (1).
move 'rewrite' to cycle_values (2).
move 'delete' to cycle_values (3).
move 'read' to cycle_values (4).
move 4 to number_values.

perform screen update with status (status_code),
    giving displaytype (11)
    cycle (cycle_values, number_values).

```

In this example, both the size and content of the cycle list are changed.

If you want the user to have the option of omitting a value for a field, you must include the null field value among the cycle values. For fields without pictures, the null value is a string of space characters. For more information on null pictures, see Chapter 3, "The Elements of FMS."

The following example modifies the display type to allow the user to omit a field value if the field is not required.

```

move spaces to cycle_values (5).
move 5 to number_values.

perform screen update with status (status_code),
    giving displaytype (11)
    cycle (cycle_values, number_values).

```

Note that if a field is required, the user must give a non-null value.

Indexed Cycle Lists

Often it is convenient within the application program to reference cycle values as an integer index into the cycle list. This is especially true if you use a table to hold the cycle values. If the `INDEXED_CYCLE_LIST` display-type action switch is true for a cycle field, then the value returned to the field-value variable is not the value that appears in the field, but an integer indicating the position of that value in the cycle list. The index is zero-based; this means that if the cycle list has N values, the index returned is in the range 0 to $N-1$, inclusive.

The following example uses a field with an indexed cycle list.

```

%replace INDEXED_CYCLE_LIST by 2048
copy 'emp_record_ids.incl.cobol'.

01 cycle_values.
02 cycle_table          pic x(12) display-2 occurs 3 times.

01 display_types.
02 display_type_value   comp-4 occurs emp_record_max_ids times.

01 emp_struct.
02 operation            comp-4.
   .
   .
   .

/* Initialize screen and create a new display type. */

move 'write' to cycle_table (1).
move 'rewrite' to cycle_table (2).
move 'delete' to cycle_table (3).

perform screen initialization 'emp_record'
       with displaytypes (display_types) status (status_code),
       giving displaytype (11) action (INDEXED_CYCLE_LIST)
       cycle (cycle_values).

/* Move the new display type to the operation field. */

move 11 to display_type_value (operation_id).

/* Display the form and get input from the user. */

perform screen input 'emp_record' update (emp_struct)
       with displaytypes (display_types) status (status_code).

/* Value specified by the user in the operation
   field is cycle_table (operation + 1). */
   .
   .
   .

```

In this example, the cycle values for the operation field are three character strings. But because the INDEXED_CYCLE_LIST action switch is set for the field, the value returned into the operation field-value variable by the perform screen input statement is an integer in the range 0 to 2, inclusive. Note that operation is defined as comp-4. Note also that the returned index value must be incremented by one to correctly subscript the table.

Obtaining Cycle List Values

You can use the `cycle_array` option in a display-type description within a `perform screen inquire` statement to return the cycle list for a previously defined cycle display type. The `cycle_array` option returns a table of cycle values and an integer indicating the number of cycle values in the table. You can use this option to obtain information about any display type: global, predefined, or temporary.

```
perform screen inquire with status (status_code),
    giving displaytype (11)
    cycle_array (cycle_values, number_values).
```

The table you specify in the `cycle_array` option — `cycle_values` in the example — must be large enough to hold all the cycle values for the display type. The component data type of the table must be convertible from a string data type.

Display-Type Option Reference Guide

This section describes each display-type option and explains how to use it. The display-type options are given in alphabetical order.

► `action ({ action_switches [, [^] update_switches] ... } , [^] update_switches ...)`

The `action` option specifies action attributes for the display type.

The operands *action_switches* and *update_switches* must be comp-5 values. In the `perform screen inquire` statement, *action_switches* must be a reference to a comp-5 variable.

The operand *action_switches* is required when defining a field and in the `perform screen inquire` statement. In the `perform screen inquire` statement, *update_switches* is not allowed and the `action` option is output-only. In all other contexts, the `action` option is input-only.

The action switches for the display type are encoded in *action_switches* and in *update_switches*. The following table shows how the switches are encoded.

Bit	Switch Name
1	AUTO_TAB_TO_NEXT_FIELD
2	TRAP_ON_FIELD_EXIT
4	TRAP_ON_FIELD_ENTRY
8	FORCE_INSERT_MODE
16	FORCE_OVERLAY_MODE
1024	BANK_TELLER_DECIMAL
2048	INDEXED_CYCLE_LIST
2^{28}	SPECIAL_ACTION_28
2^{29}	SPECIAL_ACTION_29
2^{30}	SPECIAL_ACTION_30

All unused bits are reserved. Reserved bits must be set to 0 on input and should be ignored on output.

The include file

(master_disk)>system>include_library>form_displaytype.incl.cobol defines mnemonic constants for the action switches.

If you omit the action option when defining a display type, all action switches are set to false.

You can specify the action switches in two parts: a base value and a series of update values to be applied to that base.

If you specify *action_switches*, it is taken as the base value for the action switches. If you do not specify *action_switches*, the current settings of the action switches for the display type are taken as the base value.

Each *update_switches* specifies switches to be turned on or off. If the *update_switches* value is not preceded by a circumflex (^), any action switches that are true in *update_switches* are turned on (set to true) for the display type. If the *update_switches* value is preceded by a circumflex (^), any action switches that are true in *update_switches* are turned off (set to false) for the display type.

If an *update_switches* specifies that a switch that is already true should be set to true, or that a switch that is already false should be set to false, then the switch is not changed.

The explanation of each action switch follows.

AUTO_TAB_TO_NEXT_FIELD

If this switch is true, the field is exited as soon as the user types a complete value in the field. Normally, when a field is exited, the cursor moves to the next field. In this context, a *complete value* refers to a value that fills all the positions in the field.

If you set both the **AUTO_TAB_TO_NEXT_FIELD** and **TRAP_ON_FIELD_EXIT** switches to true, control returns to the application as soon as the user gives a complete value for the field.

TRAP_ON_FIELD_EXIT

If this switch is true, control returns to the application when the user attempts to move the cursor out of the field. The action of returning control to the application is called a *trap*.

A trap on field exit is also referred to as an *immediate return*.

If you supply the **keyused** form option in a **perform** screen input statement, then the value -2 is returned in that option when a trap on field exit occurs. If you supply the **nextcursor** form option, the cursor position returned is the position to which the cursor would have moved if the trap had not occurred.

When a trap occurs on field exit, the full field validation sequence is performed. A validation routine (if any) specified for the field is invoked, and any error message is returned to the user. Other fields in the form might or might not be validated, depending on the settings of the **VALIDATE_ERRORS_OFF** and **VALIDATE_ONE_FIELD** switches of the **options** form option. The **options** form option is described in Chapter 5, "Form Options."

For more information on trapping, see Chapter 13, "Traps."

TRAP_ON_FIELD_ENTRY

If this switch is true, control returns to the application when the user attempts to move the cursor into the field. The action of returning control to the application is called a *trap*.

If you supply the `keyused` form option in a `perform screen input` statement, then the value `-9` is returned in that option when a trap on field entry occurs. If you supply the `nextcursor` form option, the cursor position returned is the position where the cursor would have been if the trap had not occurred.

When a trap occurs on field entry, the full field validation sequence is performed for all fields.

For more information on trapping, see Chapter 13, “Traps.”

FORCE_INSERT_MODE

If this switch is true, the initial edit mode for the field is insert mode. The user can change the edit mode by pressing the `(INSERT/OVERLAY)` key.

If you do not specify `FORCE_INSERT_MODE` or `FORCE_OVERLAY_MODE`, the initial edit mode is determined by the device driver.

FORCE_OVERLAY_MODE

If this switch is true, the initial edit mode for the field is overlay mode. The user can change the edit mode by pressing the `(INSERT/OVERLAY)` key.

If you do not specify `FORCE_INSERT_MODE` or `FORCE_OVERLAY_MODE`, the initial edit mode is determined by the device driver.

BANK_TELLER_DECIMAL

This switch is meaningful only if the field has a numeric picture that includes a decimal point. If this switch is true, the field behaves in a way common to many bank-teller machines. The cursor appears to the right of the field. Digits typed in the field appear first at the right edge of the field (to the right of the decimal point) and move left as additional digits are typed. The digits 0 through 9 are the only characters a user can type in a bank-teller decimal field.

For example, if the field picture is `z.99`, the field value initially appears as `.00`. If the user types the digits `123`, the field value changes to `.01`, to `.12`, and finally to `1.23`.

If the `BANK_TELLER_DECIMAL` switch is false for a field with a numeric picture that contains a decimal point, the cursor initially appears at the decimal point. Digits typed in the field initially appear to the left of the decimal point and move left as additional characters are typed.

The cursor does not move. When the user types a decimal point, the content of the field does not change, but the cursor moves to the right of the decimal point. Any additional characters typed by the user appear to the right of the decimal point, and the cursor moves right as each additional character is typed.

Note: Some device drivers do not support bank-teller decimal.

INDEXED_CYCLE_LIST

This switch is only meaningful for cycle fields (see the description of the cycle display-type option later in this chapter). If this switch is true, the value of the cycle field is not stored in the field-value variable. Instead, the field-value variable stores an index into the list of cycle values. The first value in the cycle list corresponds to index value 0, the second value to index value 1, and so forth.

If the INDEXED_CYCLE_LIST switch is true for a field, the data type of the associated field-value variable must be comp-4.

Note that using the INDEXED_CYCLE_LIST does not alter the field's appearance. Only the value stored in the field-value variable changes.

SPECIAL_ACTION_28

SPECIAL_ACTION_29

SPECIAL_ACTION_30

Specific device drivers might assign meaning to these switches. The values of these switches are ignored by the Forms Processor.

► charset (*character_set_id*)

The charset display-type option specifies the only character set supported by the field.

The operand *character_set_id* must be a comp-4 value. The charset option is output-only in the perform screen inquire statement. In all other contexts, it is input-only.

The value *character_set_id* indicates which character set is supported by the field. The following table lists the allowed values and their meanings.

Character Set ID	Character Set Name
0	ASCII_CHARACTER_SET
1	LATIN_1_CHAR_SET
2	KANJI_CHAR_SET
3	KATAKANA_CHAR_SET
4	HANGUL_CHAR_SET

Constants for these values are defined in the file
(master_disk)>system>include_library>char_sets.incl.cobol.

If you specify a charset option, any characters in the field value are assumed to be of the specified character set. If the user attempts to submit a value that contains characters from another character set, the form is not accepted.

For information on supplemental character sets, see the section “International Character Set Support,” in Chapter 3, “The Elements of FMS.”

You can specify the manner in which supplemental characters are stored in the field-value variable by using the shift and unshift field options described in Chapter 6, “Field Descriptions.”

► cycle ({ $\begin{matrix} cycle_value [, cycle_value] \dots \\ cycle_value_array [, value_count] \end{matrix}$ })

The cycle display-type option specifies a cycle list for a display type.

If you specify *cycle_value* operands, each must be a valid value for the field to which the display type is applied. If you use the operand *cycle_value_array*, it must be a reference to a table of valid values for the field, and *value_count* must be a comp-4 value. All the operands are input-only.

The two syntaxes of the `cycle` display-type option allow you to specify the cycle list in two ways:

- by listing the cycle values, using the *cycle_value* operands
- by specifying a table of cycle values (*cycle_value_array*) and an integer indicating the number of cycle values in that table (*value_count*).

Using the second syntax, you can change the cycle values by changing the values in *cycle_value_array*. You can also change the number of cycle values by changing the value of *value_count*. Only the first *value_count* elements of the table are used.

If you use a variable for each *cycle_value* in the first syntax, you cannot change the number of cycle values, but you can change the cycle values themselves by changing the values of those variables.

You can use the `cycle_array` display-type option to return the current cycle list for a display type.

► `cycle_array (cycle_value_array, value_count_variable)`

The `cycle_array` display-type option returns the cycle list for a cycle display type.

The operand *cycle_value_array* must be a reference to a table, and *value_count_variable* must be a reference to a `comp-4` variable. Both operands are output-only. The component data type of *cycle_value_array* must be compatible with the cycle values.

You can use the `cycle_array` option only in the `perform screen inquire` statement.

The number of values in the cycle list is returned in *value_count_variable*. The size of *cycle_value_array* must be at least large enough to hold that many values. The cycle values are returned in the first *N* elements of the table, where *N* is the number returned in *value_count_variable*.

You can use the `cycle` display-type option to set or change the cycle list for a display type.

► **picture** (*picture*)

The **picture** display-type option specifies a field picture, or template, for a display type.

The operand *field_picture* must be convertible to a character string. In the **perform screen inquire** statement, the **picture** display-type option is output-only. In all other contexts, it is input-only.

Each character in *field_picture* corresponds to a character position in a field and defines the valid characters for that field position. If *field_picture* is shorter than the associated field, the Forms Processor automatically extends the field picture to the length of the field. For more information, see Chapter 9, “Field Pictures and Filtering.”

The following table lists the valid picture characters.

Picture Character	Meaning
A or a	Allow space or hyphen.
B or b	Insert literal space.
L or l	Allow letter, digit, or space; convert letter to lowercase.
U or u	Allow letter, digit, or space; convert letter to uppercase.
X or x	Allow letter, digit, or space.
Z or z	Allow digit or hyphen (negative sign); suppress leading zeros.
9	Allow digit or hyphen (negative sign); display leading zeros.
.	Fix decimal point location. [†]
,	Fix digit-grouping character location. [†]
-	Insert literal hyphen.
/	Insert literal slant.

[†] The meaning of the period and comma can be reversed with the **decimal is comma** option in the Forms Editor (**MENU**) **S** form.

For information on field pictures, see Chapter 9, “Field Pictures and Filtering.”

► range ({ $\begin{matrix} min_value & [,max_value] \\ & ,max_value \end{matrix}$ }),

The range display-type option specifies a minimum and maximum field value for a display-type.

The operands *min_value* and *max_value* must each be convertible to the data type of the field-value variable for the associated field. In the perform screen inquire statement, both values are output-only. In all other contexts, both values are input-only.

The range display-type option restricts field values by specifying a range of valid values. When the user submits the form, the Forms Processor checks that the input value is within the specified bounds. If the value is out of range, the Forms Processor displays an error message and positions the cursor to the field.

If you omit *min_value*, the range option establishes an upper bound for the field, but does not establish a lower bound. Similarly, if you omit *max_value*, the range option establishes a lower bound for the field, but does not establish an upper bound.

The Forms Processor performs the range test after checking that the value conforms to the field picture (if any) and before calling the field validation routine (if any).

► validate (*validation_entry*)

The validate display-type option establishes a validation routine for a display type.

The operand *validation_entry* must be a reference to an entry value or a character-string containing the name of a validation routine.

The Forms Processor invokes the validation routine to check the field value when the user submits the form.

When the form is submitted, the Forms Processor first checks that the field value conforms to the field picture (if any). It then checks that the value is within the range specified for the field (if any). It then invokes the validation routine (if any).

For information on writing validation routines, see Chapter 10, "Error Handling and Field Validation."

► visual ({ *visual_switches* [, [^] *update_switches*] ... } , [^] *update_switches* ...)

The visual display-type option specifies visual attributes of the display type.

The operands *visual_switches* and *update_switches* must be convertible to comp-5 values. In the perform screen inquire statement, *visual_switches* must be a reference to a comp-5 variable.

The operand *visual_switches* is required when defining a field and in the perform screen inquire statement. In the perform screen inquire statement, *update_switches* is not allowed and the visual option is output-only. In all other contexts, the visual option is input-only.

The visual switches for the display type are encoded in *visual_switches* and in *update_switches*. The following table shows how the switches are encoded.

Bit	Switch Name
1	BLANKED_VISUAL
2	BLINKING_VISUAL
4	INVERSE_VISUAL
8	UNDERLINED_VISUAL
16	LOW_INTENSITY_VISUAL
32	HIGH_INTENSITY_VISUAL
256	CENTER_FIELD_DATA
512	RIGHT_JUSTIFY_FIELD_DATA
1024	LEFT_JUSTIFY_FIELD_DATA
2048	NOTRIM_FIELD_DATA_SPACES
2 ²⁸	SPECIAL_VISUAL_28
2 ²⁹	SPECIAL_VISUAL_29
2 ³⁰	SPECIAL_VISUAL_30

All unused bits are reserved. Reserved bits must be set to 0 on input and should be ignored on output.

The include file

(master_disk)>system>include_library>form_displaytype.incl.cobol defines mnemonic constants for the visual switches.

If you omit the visual option when defining a display type, all visual switches are set to false.

You can specify the visual switches in two parts: a base value and a series of update values to be applied to that base.

If you specify *visual_switches*, it is taken as the base value for the visual switches. If you do not specify *visual_switches*, the current settings of the visual switches for the display type are taken as the base value.

Each *update_switches* specifies switches to be turned on or off. If the *update_switches* value is not preceded by a circumflex (^), any visual switches that are true in *update_switches* are turned on (set to true) for the display type. If the *update_switches* value is preceded by a circumflex (^), any visual switches that are true in *update_switches* are turned off (set to false) for the display type.

If an *update_switches* specifies that a switch that is already true should be set to true, or that a switch that is already false should be set to false, then the switch is not changed.

Note: Visual attributes are highly device dependent. Some of the visual switches might not be supported for some devices. Other visual switches might behave somewhat differently for different devices.

The explanation of each visual switch follows.

BLANKED_VISUAL

If this switch is true, no visible text appears in the field. If the field is an input field, the user can type in the field, but the characters are not displayed. However, the value given is returned to the application in the field-value variable. This switch is useful for password fields.

BLINKING_VISUAL

If this switch is true, all text in the field blinks on and off.

INVERSE_VISUAL

If this switch is true, the field is in reverse video. This means that if the terminal normally displays light characters on a dark background, this field displays dark characters on a light background. Similarly, if the terminal normally displays dark characters on a light background, this field displays light characters on a dark background.

UNDERLINED_VISUAL

If this switch is true, the entire field is underlined.

LOW_INTENSITY_VISUAL

If this switch is true, text within the field appears at low intensity. This switch and the *HIGH_INTENSITY_VISUAL* switch are mutually exclusive.

HIGH_INTENSITY_VISUAL

If this switch is true, text within the field appears at high intensity. This switch and the *LOW_INTENSITY_VISUAL* switch are mutually exclusive.

CENTER_FIELD_DATA

If this switch is true, output values are centered in the field. For most device drivers, you cannot set this switch to true for input fields.

This switch, the **RIGHT_JUSTIFY_FIELD_DATA** switch, and the **LEFT_JUSTIFY_FIELD_DATA** switch are mutually exclusive.

RIGHT_JUSTIFY_FIELD_DATA

If this switch is true, text in the field is right-justified. This switch, the **CENTER_FIELD_DATA** switch, and the **LEFT_JUSTIFY_FIELD_DATA** switch are mutually exclusive.

If an input field is right-justified, the cursor appears at the field's right edge when the user positions to the field. Each character the user types initially appears in the rightmost character position. As subsequent characters are typed, the characters in the field move to the left. The cursor never moves.

LEFT_JUSTIFY_FIELD_DATA

If this switch is true, text in the field is left-justified. This switch, the **CENTER_FIELD_DATA** switch, and the **RIGHT_JUSTIFY_FIELD_DATA** switch are mutually exclusive.

If an input field is left-justified, the cursor appears at the field's left edge when the user positions to the field. The first character typed by the user appears in the leftmost character position. As each character is typed, the cursor moves one position to the right. The second character typed appears in the second position from the left, and so forth.

NOTRIM_FIELD_DATA_SPACES

If this switch is true, space characters are not removed from the left or right of a string output value before that value is displayed in the form. Similarly, space characters are not trimmed from an input value before that value is stored in a string field-value variable.

SPECIAL_VISUAL_28

SPECIAL_VISUAL_29

SPECIAL_VISUAL_30

Specific device drivers might assign meaning to these switches. The values of these switches are ignored by the Forms Processor.

Chapter 8:

Data States

This chapter describes field data states. It discusses how data states are initialized in the Forms Editor and how they can be read and altered by an application program.

The Data-State Switches

Each field in a form has an associated set of 16 switches that defines the data state of the field. Some of the data-state switches are output switches: they return information about the field. Other switches are input-output: you can alter the value of these switches to change the field.

Within an FMS program, you can store the data-state switches in a two-byte integer value. Table 8-1 lists the data-state switches that are currently defined.

Table 8-1. The Data-State Switches

Bit	Switch Name	Type
1	DISAPPEARING_DEFAULT	(input-output)
2	FIELD_VALUE_GIVEN	(output)
4	FIELD_HAS_CHANGED	(input-output)
16	REQUIRED_FIELD	(input-output)
32	INPUT_FIELD	(input-output)
64	NEW_DATA_IN_FIELD	(input-output)
128	DISABLE_ENTIRE_FIELD	(input-output)
256	FILTER_FOR_CONVERSION	(output)

Each switch is either true or false. The meaning of each switch is described later in this chapter under the heading "Data State Reference Guide."

Data States in the Forms Editor

You can initialize some of the data-state switches for each field in the Forms Editor (MENU) F form. The table lists data-state switches that are initialized by (MENU) F options.

Data-State Switch	(MENU) F Option
DISABLE_ENTIRE_FIELD	DISABLE
DISAPPEARING_DEFAULT	DISAPPEARING
INPUT_FIELD	FIELD TYPE
REQUIRED_FIELD	REQUIRED

The DISAPPEARING, REQUIRED, and DISABLE fields of the (MENU) F form each cycles to yes or no. A yes value initializes the corresponding data-state switch to true. A no value initializes the corresponding data-state switch to false.

The FIELD TYPE field cycles to input, output, or output only. If you specify input, the INPUT_FIELD switch is initialized to true. If you specify output, the INPUT_FIELD switch is initialized to false. In either of these cases, you can change the field type within the program by changing the value of the INPUT_FIELD switch. If you specify output only for FIELD TYPE in the Forms Editor, the INPUT_FIELD switch is initialized to false, but the effect of changing that value is undefined. If you want to change the field type dynamically, you should always set FIELD TYPE to either input or output.

Referencing Data States in a Program

Within a program, you can reference the data states of all fields in a form with the `datastates` form option, or you can reference the data state of a specific field with the `datastate` field option.

If you do not use the `datastates` form option or the `datastate` field option for the form within the program, the data-state values specified by the Forms Editor are used.

Data-State Variables

The data state for a field can be stored in a `comp-4` variable. For example, you could declare a data-state variable as follows:

```
01 data_state    comp-4.
```

You can access an individual switch within the `data_state` variable mathematically using the following mnemonic constants defined in the file `(master_disk)>system>include_library>form_datastate.incl.cobol`.

```
%replace FILTER_FOR_CONVERSION by 256
%replace DISABLE_ENTIRE_FIELD by 128
%replace NEW_DATA_IN_FIELD by 64
%replace INPUT_FIELD by 32
%replace REQUIRED_FIELD by 16
%replace FIELD_HAS_CHANGED by 4
%replace FIELD_VALUE_GIVEN by 2
%replace DISAPPEARING_DEFAULT by 1
```

For example, you can test the `REQUIRED_FIELD` switch as follows:

```
if data_state >= 0 then
    compute quotient = data_state / REQUIRED_FIELD
else compute quotient = (data_state + REQUIRED_FIELD - 1) /
    REQUIRED_FIELD.

if !mod (quotient, 2) equal 1 then
    /* Field is required. */
else /* Field is not required. */.
```

After you have determined that the `REQUIRED_FIELD` switch is false, you can set it to true with the following assignment:

```
add REQUIRED_FIELD to data_state.
```

If the `REQUIRED_FIELD` switch is true, you can set it to false with the following assignment:

```
subtract REQUIRED_FIELD from data_state.
```

To access the data states of all fields in a form, you can declare a table of data-state variables as follows:

```
01 data_states.
    02 data_state_value comp-4 occurs form_name_max_ids.
```

The value `form_name_max_ids` is defined in the field-IDs include file produced by the Forms Editor. For more information on this file, see Chapter 4, "The Forms Editor."

You can reference the data state of a single field from the `data_states` table as follows:

```
data_state_value (field_name_id)
```

Definitions of *field_name_id* for each field are included in the field-IDs include file.

Reading Data States

You can use the *datastate* field option to read the data state of a single field, or you can use the *datastates* form option to read the data state of all fields of a form.

Single Field. You can initialize the data-state variable for a single field by referencing that variable in the *datastate* option in a field description clause of a perform screen initialization statement. The following example initializes the variable *field_data_state* to the data state of the *name* field.

```
perform screen initialization 'menu_form'
    with status (error_code),
    using field (name_id) datastate (field_data_state).
```

The example assumes that the program includes the field-IDs file that defines *name_id*. The perform screen initialization statement initializes the value of *field_data_state* to the switch values specified for the *name* field in the Forms Editor. The variable *field_data_state* should be declared as *comp-4*.

You can also use the *datastate* option in a field description clause of a perform screen input, perform screen output, perform screen update, or perform screen inquire statement to obtain the current data state of a field. The following example displays the form, accepts input from the user, and then updates the value of *field_data_state* to the current data state of the *name* field.

```
perform screen input with status (error_code),
    using field (name_id) datastate (field_data_state).
```

Note that the value of *field_data_state* is updated after the form is submitted. Therefore, the value of *FIELD_VALUE_GIVEN*, *FIELD_HAS_CHANGED*, and other output switches reflect activity from this display of the form.

All Fields. You can initialize a table of data-state variables to the field data states defined in the Forms Editor by referencing that table in the *datastates* option of the perform screen initialization statement. The following example initializes *data_states_array*.

```
perform screen initialization 'menu_form'
    with datastates (data_states_array) status (error_code).
```

The elements of *data_states_array* should be *comp-4* variables. The data state for the field whose field ID is 1 is contained in the first element of *data_states_array*. The data state for the field whose field ID is 2 is in the second element, and so forth.

You can also use the *datastates* option in a perform screen input, perform screen output, perform screen update, or perform screen inquire statement to obtain the current

data states of all fields in a form. The following example displays the form and then updates the values in `data_states_array` to the current data state of the fields.

```
perform screen output with datastates (datastates_array) status (error_code).
```

As with the `datastate` option, the operand of the `datastates` option is updated after the form is submitted. The output data-state switches reflect activity from this display of the form.

Changing Data States

Normally, the `datastates` form option and the `datastate` field option are output-only options. However, you can make them input-output options in a `perform screen input`, `perform screen output`, or `perform screen update` statement by turning on the `COPY_DATASTATE` option switch. This allows you to specify a new data state for one or more fields within the application program. For information on the `COPY_DATASTATE` switch, see the description of the options form option in Chapter 5, "Form Options."

Within a program, you can alter the following data-state switches for a field:

```
DISAPPEARING_DEFAULT
FIELD_HAS_CHANGED
REQUIRED_FIELD
INPUT_FIELD
NEW_DATA_IN_FIELD
DISABLE_ENTIRE_FIELD
```

If you set the `FIELD_HAS_CHANGED` or `NEW_DATA_IN_FIELD` switch to true for a form display, the display is not affected, but on return from the display, the switch is still true. The other input data-state switches affect the display of the form.

You can alter a data-state value read in a previous statement, or you can specify an entirely new data-state value.

For example, if you read the data states for a form into a table of `comp-4` variables, `data_state_array`, after determining that the `REQUIRED_FIELD` switch for a field is false, you can set it to true as follows:

```
add REQUIRED_FIELD to data_state_array (field_id).
```

The example assumes that the program includes the system include file `(master_disk)>system>include_library>form_datastate.incl.cobol`.

You can subsequently set the `REQUIRED_FIELD` switch to false as follows:

```
subtract REQUIRED_FIELD from data_state_array (field_id).
```

The changes you make to a data-state variable take effect on the field only if you reference the data-state variable in a perform screen input, perform screen output, or perform screen update statement with the COPY_DATASTATE option true.

```
perform screen input 'menu_form' update (field_values)
    with datastates (data_state_struct) options (COPY_DATASTATE)
    status (error_code).
```

The example assumes that the program includes the system include file (master_disk)>system>include_library>form_datastate.incl.cobol. For information on the options form option, see Chapter 5, “Form Options.”

Data State Reference Guide

In this section, the data-state switches are described in alphabetical order.

DISABLE_ENTIRE_FIELD (input-output)

If this switch is true, the field is invisible to the user. The field is not displayed and the user cannot position the cursor to the field. By altering the value of this switch within a program, you can make a field appear on some displays of a form and not appear on others.

While a field is disabled, the field remains allocated in the display list and can be re-enabled at any time. To remove a field from the display list, use the perform screen delete statement described in Chapter 16, “Statements.”

The DISABLE_ENTIRE_FIELD switch is initially true for uncommitted fields and initially false for all other fields.

DISAPPEARING_DEFAULT (input-output)

If this switch is true, then the initial output value for the field disappears when the user begins typing at the start of the field. Such fields are sometimes referred to as *non-editable*. The user can edit the default value by positioning the cursor beyond the first character of the field before entering or deleting any characters. The DISAPPEARING_DEFAULT switch has no effect on the user’s ability to correct typing mistakes or otherwise change a typed value.

For predefined fields, the DISAPPEARING switch in the (MENU) F form establishes an initial value for the DISAPPEARING_DEFAULT switch.

FIELD_HAS_CHANGED (input-output)

If this switch is true, the value of the field was changed by the most recent perform screen input or accept statement for which the COPY_DATASTATE switch was false. For information on the COPY_DATASTATE switch, see the description of the options form option in Chapter 5, "Form Options."

If the COPY_DATASTATE switch is true for a perform screen input or accept statement, then the Forms Processor does not reset the value of the FIELD_HAS_CHANGED switch. If the COPY_DATASTATE switch is false for a perform screen input or accept statement, then the FIELD_HAS_CHANGED switch is output-only. In the latter case, the Forms Processor sets the FIELD_HAS_CHANGED switch to false before displaying the form, and updates the switch after the form display.

Note: Some device drivers might set this switch more often than necessary.

FIELD_VALUE_GIVEN (output)

If this switch is true, a non-null value was specified for the field in the most recent perform screen input or accept statement that displayed the form. The field value might be a value typed by the user, or it might be an initial output value.

This switch is meaningful only after the form is displayed with the perform screen input or accept statement.

FILTER_FOR_CONVERSION (output)

If this switch is true for a perform screen initialization or accept statement, then the value of the field might be filtered before being converted and stored in the field-value variable. To filter a value means to remove non-numeric characters from a numeric value and transform it into a standard representation.

The characters that must be filtered from a numeric value include the currency character (\$) by default, some sign characters, and zero-suppression characters. For more information on filtering, see Chapter 9, "Field Pictures and Filtering."

The Forms Processor determines which fields might require filtering, based on the picture defined for the field. You cannot directly change the value of the FILTER_FOR_CONVERSION switch.

INPUT_FIELD (input-output)

If this switch is true, the field is an input field. This means that when the form is displayed with the `perform screen input` or `accept` statement, the user can position to the field. The user can also modify the field value, unless the `WIDE_CURSOR` option switch is true. For information on the `WIDE_CURSOR` switch, see the description of the options form option in Chapter 5, "Form Options."

For a predefined field, the `FIELD TYPE` option of the Forms Editor (**MENU**) `F` form determines the initial value for the `INPUT_FIELD` switch. If the `FIELD TYPE` value is `input`, the `INPUT_FIELD` switch is true; otherwise, it is false. If the `FIELD TYPE` value is `output only`, you must not change the value of the `INPUT_FIELD` switch within the program. If the field type is either `input` or `output`, you can change the value of the `INPUT_FIELD` switch to dynamically change the field type.

NEW_DATA_IN_FIELD (input-output)

The Forms Processor sets this switch to true whenever a field becomes empty (that is, it takes on its null value). For example, if the user deletes the entire field value before typing a new value, the `NEW_DATA_IN_FIELD` switch is set to true. If the user modifies the initial value rather than deleting it, the `NEW_DATA_IN_FIELD` switch is not altered.

If the `COPY_DATASTATE` switch is false for the `perform screen input` or `accept` statement that displays the form, the Forms Processor initializes the `NEW_DATA_IN_FIELD` switch to false before displaying the form. If the `COPY_DATASTATE` switch is true, the Forms Processor takes the current value of the `NEW_DATA_IN_FIELD` switch as the initial value. For information on the `COPY_DATASTATE` switch, see the description of the options form option in Chapter 5, "Form Options."

The `NEW_DATA_IN_FIELD` switch is useful for applications that must treat fresh field values differently than modified field values. For example, the application might expand abbreviations within a fresh value.

REQUIRED_FIELD (input-output)

If this switch is true, the user cannot submit the form without specifying a non-null value for this field. For information on null field values, see Chapter 3, "The Elements of FMS."

If the user attempts to submit a form that has a null value for a required field, the terminal bell sounds, the cursor is positioned to the required field, and the following message is displayed:

A required field is missing.

For a predefined field, the value of the REQUIRED field in the Forms Editor (MENU) F form determines the initial value of the REQUIRED_FIELD switch.

Note: Some programmers prefer to set the REQUIRED_FIELD switch for each field to false and use a field validation routine to ensure that a valid value is specified for a field. For information on field validation routines, see Chapter 10, "Error Handling and Field Validation."

Chapter 9:

Field Pictures and Filtering

This chapter discusses field pictures that you can specify in the Forms Editor or in the picture display-type option of a screen statement. It also discusses filtering: the process of translating between a character-string representation of a number and a true numeric value.

Field Pictures

A *field picture* is a string of characters that indicates what characters are valid in each position of a field. The field picture is always the same length as the field it describes. (If you specify a shorter field picture, the Forms Processor automatically extends the picture to the length of the field.) The first character of the picture indicates what characters are valid for the first character position in the field. The second character of the picture indicates what characters are valid for the second position in the field, and so forth.

Table 9-1 lists the characters that you can include in a field picture and the type and meaning of each.

Table 9-1. The Field Picture Characters

Picture Character	Type	Meaning
,	Numeric	Group digits; or fix location of decimal point.
-	Alphanumeric	Insert literal hyphen.
.	Numeric	Fix location of decimal point; or group digits.
/	Alphanumeric	Insert literal slant.
9	Numeric or alphanumeric	Allow a digit. In a numeric picture, also allow a sign.
A a	Alphanumeric	Allow a letter or a space character.
B b	Alphanumeric	Insert literal space character.
L l	Alphanumeric	Allow a letter, digit, or space character; convert a letter to lowercase.
U u	Alphanumeric	Allow a letter, digit, or space character; convert a letter to uppercase.
X x	Alphanumeric	Allow a letter, digit, or space character.
Z z	Numeric	Allow a digit or sign; suppress leading zeros.

The uppercase and lowercase versions of a picture character are interchangeable.

The picture characters are explained in more detail later in this chapter.

Note that some picture characters are numeric and others are alphanumeric. A field picture must contain all numeric characters or all alphanumeric characters. For example, a picture cannot contain both the X character and the Z character. Note that the 9 character can be either numeric or alphanumeric.

A picture consisting of numeric characters is a *numeric picture*. A picture consisting of alphanumeric characters is an *alphanumeric picture*. Each of these two types of picture is discussed later in this chapter.

Specifying a Picture

You do not have to specify a field picture for every field in a form. The field picture is an optional means of restricting field values. Using a picture can reduce errors by preventing the user from typing invalid values.

You can specify a picture for a field in the PICTURE option of the **(MENU)** F form of the Forms Editor. You can also specify a field picture in the picture display-type

option in a screen statement. For information on the Forms Editor, see Chapter 4, "The Forms Editor." For information on the picture display-type option, see Chapter 7, "Display Types."

Alphanumeric Pictures

The alphanumeric picture characters can be divided into two groups: those that indicate a specific literal character to be inserted into a position of the field, and those that indicate a set of valid characters for a position of the field.

The hyphen (-), slant (/), and blank (B or b) picture characters specify literal characters to be inserted into the field. These literals appear in the specified position whenever the field is displayed. The user cannot type any other character in that position.

The other alphanumeric picture characters specify a set of valid characters that the user can type in the position within the field. If no character is in the specified position when the field is displayed, that character position is blank. The user can type any character from the set of valid characters in that position.

You can use the alphanumeric picture characters, for example, to establish the format for a telephone number ('999-9999') or a date ('99/99/99' or '99buaab9999').

Numeric Pictures

The 9 and z numeric picture characters represent digits of precision within a value. The period and comma picture characters indicate the location of the decimal point, and group the digits to make the value easier to read.

Note: The z picture character is valid only to the left of the decimal point and to the left of any 9 picture characters. The 9 picture character can appear anywhere within a numeric field, except to the left of a z picture character.

Field Precision. The decimal precision of a field with a numeric picture is the total number of 9 and z characters. The scaling factor of the field is the number of 9 characters to the right of the decimal point. For example, a field with the picture 'zz,zz9.99' has a decimal precision of 7 and a scaling factor of 2. To prevent overflow of the variable, the field-value variable for a numeric field should have at least as great a precision and scaling factor as the field picture.

Periods and Commas. By default, the period picture character represents the position of a decimal point within a numeric field, and the comma is a grouping character. However, you can switch the meanings of these characters if you wish.

The Forms Editor (MENU) S form includes a field that you can cycle to either decimal is period or decimal is comma. This field determines the meaning of the period and comma characters in all pictures given within the (MENU) F form. If the value is decimal is period (the default), then the period picture character marks the location of a decimal point, and the comma picture character is used to separate groups of digits. If the value is decimal is comma, the meanings of the period and comma picture characters are reversed. Note that the decimal is comma option applies only to field pictures you specify in the Forms Editor; it does not affect field pictures you specify in a program.

In VOS COBOL programs, you can switch the meanings of the decimal point and comma characters by specifying the decimal-point is comma clause in the special-names paragraph of a program. For information on this clause, see the *VOS COBOL Language Manual (R010)*.

Note: If you decide to use the comma is decimal option, issue the (MENU) S request and set that option before defining any fields. This ensures that all field pictures are interpreted correctly and consistently.

Grouping characters appear in a field only when the current field value includes digits to the left of the grouping character. Otherwise, a space character appears in place of the grouping character. For example, if the field picture is 'zz,zzz' and the current value is 123, the field value is displayed as 123. If the current value of the field is 1234, then the field value is displayed as 1,234.

If the period is used to mark the location of the decimal point within a field picture, then when the field is displayed, a period appears at that location within the field. For example, if the picture for a field is 'z,zzz.99', then the null value of that field is .00.

If a comma is used to mark the location of the decimal point within a field picture, then when the field is displayed, a comma appears at that location within the field. For example, if the picture for a field is 'zz.zzz,99', then the null value of that field is ,00.

Filtering

Filtering is the process of removing certain special characters from a value to produce a standard intermediate representation of a numeric value.

The Forms Processor must perform two types of conversion on numeric data.

- On output, before a value can be displayed in a field with a numeric picture, the Forms Processor must convert the value to a representation that is compatible with the field picture.

- On input, before a field value can be stored in a numeric field-value variable, the Forms Processor must convert the field value to a numeric value of the appropriate data type.

Each of these conversions is a two-step procedure. In the first step, called *filtering*, the Forms Processor removes certain special characters from the original value. This produces a standard representation that serves as an intermediate value. In the second step, the Forms Processor either edits the intermediate value to fit the picture of the target field, or converts the intermediate value to the data type of the target variable.

For example, if the data type of a program variable is picture ('**,**9'), it might contain the value ****12. If this value is to be displayed in a field with the picture '99,999', it must be transformed. First, the Forms Processor removes the asterisks to produce the intermediate value 12. The Forms Processor then edits the intermediate value to fit the field. The value finally displayed in the field is 00,012. If this field value is subsequently returned to the same program variable, the Forms Processor removes the leading zeros and the comma to produce the intermediate value 12, which is then converted to the data type of the variable. The value assigned to the variable is ****12.

Special Numeric Characters

The following characters indicate the sign of an unfiltered value:

-, +, cr, CR, db, DB

The plus sign (+) and hyphen (-) can appear at the beginning or end of the value. The credit (cr or CR) and debit (db or DB) indicators can appear only at the end of the value. The plus sign (+) indicates a positive value. The other sign characters indicate a negative value.

In a filtered value, a leading hyphen (-) indicates a negative value. No sign indicator is used for positive values.

The following characters are valid in unfiltered values, but are not allowed in filtered values.

- The sign characters (except that a filtered value can contain a leading hyphen)
- The currency symbol (\$ is the default)
- The slant (/)
- The asterisk (*)
- The space character

These characters are called *special numeric characters*. The Forms Processor removes these characters, but maintains the sign and magnitude of the original value.

The Filtering Process

The Forms Processor performs the following steps to filter a value.

1. The Forms Processor deletes any currency symbols, grouping characters, spaces, slants, and asterisks from the value.
2. The Forms Processor removes any leading or trailing sign indicators, but keeps track of whether the number is positive or negative.
3. The Forms Processor determines the location of the decimal point.
4. The Forms Processor deletes any leading zeros.
5. If the value is negative, the Forms Processor puts a hyphen (minus sign) in front of the value.
6. If the value is now empty or consists only of a decimal point or minus sign, the Forms Processor replaces the value with a zero.

Table 9-2 lists some examples of unfiltered values and the corresponding filtered values.

Table 9-2. Examples of Filtering

Unfiltered Value	Filtered Value
123	123
\$**123db	-123
-12.3e+1	-12.3e+1
12.3E-20CR	-12.3E-20
\$ 0.123+	.123
1,23	123
09/03/90	90390
9 03 1990	9031990

Filtering Output Values

Output values transmitted to the form often require filtering when the value's data type is picture or character string. Output values are filtered only when the target field has an associated picture.

The process of transforming the program value to a field value is as follows:

1. The Forms Processor filters the value.
2. The Forms Processor edits the filtered value to conform to the field picture.

Editing the value might involve aligning the value on a decimal point within the field, adding leading and trailing zeros, and inserting grouping characters.

Filtering Input Values

Input values transmitted from the form to numeric program variables are filtered as part of the field validation suite. The validation suite for each field is as follows:

1. The Forms Processor checks that the field value conforms to the field picture.
2. If the field-value variable has a numeric or picture data type, then the Forms Processor filters the value.
3. The Forms Processor converts the filtered value to the data type of the field-value variable.
4. If the field has a range restriction, the Forms Processor checks that the converted value is within that range.
5. If you have supplied a validation routine for the field, the Forms Processor passes the **unfiltered** value to that routine.

For more information on the field validation suite and, specifically, on programmer-written validation routines, see Chapter 10, "Error Handling and Field Validation."

Chapter 10:

Error Handling and Field Validation

This chapter discusses the processes by which the Forms Processor validates fields and detects and reports errors within a forms application. This includes an explanation of how to write your own field validation routines.

Handling Forms Errors

The Forms Processor recognizes two kinds of errors, which it handles in different ways.

- If the Forms Processor detects that the user has submitted a form that contains an invalid field value, the Forms Processor redisplay the form with an appropriate error message and allows the user to correct the field value.
- If the Forms Processor encounters any other kind of error, it returns control to the program. The Forms Processor indicates the error to the program in one of two ways.
 - If the screen statement for which the error occurs includes the `status` form option, the Forms Processor returns an appropriate non-zero error code in that option.
 - If the statement does not contain the `status` option, the Forms Processor signals the error condition in the block that contains the screen statement. The Forms Processor sets the oncode value for the condition to an appropriate error code.

For information on how the Forms Processor detects and handles the first kind of error, see the section headed “Field Validation” later in this chapter.

To handle the second kind of error, the best practice is to always include the `status` form option in every screen statement within a program and check the returned value in the next statement. This allows you to explicitly handle any errors encountered in a specific screen statement.

The error codes returned in the `status` option can be divided into two categories: those that you anticipate and can handle specifically, and those that you do not anticipate.

For example, if you do not include the `keyused` option in a `perform` screen input statement, then if the user cancels the form, the error code `e$form_aborted` (1453) is returned in the `status` option. You might want to check for this specific returned value and, when it is encountered, take a particular action and continue with normal program execution.

Another class of error codes that you might want to handle explicitly are those dealing with communication line problems. The exact error codes returned are dependent on the device driver, but they might include `e$parity_error` (2916) and `e$line_hangup` (1365). When these codes are returned, some applications might loop back a limited number of times and try executing the screen statement again. A program should **not** loop forever on these errors. If the error persists, the program should execute an error-reporting routine and stop.

Other error codes might be returned because of a programming error or other unexpected situation. In this case, you cannot check for a specific code, and there is usually no specific action the program can take to correct the situation. Continuing with normal program execution might produce unexpected results. Usually, the best action is to report the error to the user and stop the program.

The following code fragment illustrates forms error code handling for a program that does not use the `keyused` form option.

```
data division.  
working-storage section.  
  
01 info_fields.  
   copy 'info'.  
  
01 e$form_aborted      comp-4.  
01 error_code         comp-4.  
01 info_form_id       comp-4.  
01 my_name            picture x(32) display-2 value 'input_info'.  
01 empty_string       picture x(32) display-2 value ''.
```

(Continued on next page)

(Continued)

```
procedure division.  
  .  
  .  
  .  
  
  perform screen input 'info' update (info_fields)  
    status (error_code).  
  
  if error_code equal e$form_aborted then  
    perform cancel-form  
  else if error_code not equal 0 then  
    perform report-error-and-abort.  
  .  
  .  
  .  
  
  cancel-form.  
  
  *   User canceled form.  
  .  
  .  
  .  
  
  report-error-and-abort.  
  
  *   Report error and abort program.  
  .  
  .  
  .  
  
  call 's$error' using error_code, my_name, empty_string.  
  go to exit-program.  
  .  
  .  
  .  
  
  exit-program.  
  
  exit program.
```

Field Validation

When the user submits a form, the Forms Processor examines the value given for each field of the form and performs a series of checks to determine if the field value conforms to any restrictions defined for that field. This series of checks is called the *validation suite*.

The order in which the fields are validated is determined by the Forms Processor. The Forms Processor performs the complete validation suite for one field before beginning the validation of another field.

The Validation Suite

The steps in the validation suite are as follows:

1. If the field's `REQUIRED_FIELD` data-state switch is true, the Forms Processor checks that the field value is non-null.
2. If the field's display type has an associated field picture, the Forms Processor checks that the field value conforms to that picture.
3. If the field-value variable is numeric, the Forms Processor filters the value.
4. The Forms Processor converts the field value to the data type of the field-value variable.
5. If the field's display type has a range restriction, the Forms Processor checks that the field value is within the specified range.
6. If the field's display type has a programmer-defined validation routine, the Forms Processor invokes that routine and passes it the **unfiltered** field value.

For information on the `REQUIRED_FIELD` data-state switch, see Chapter 8, "Data States." For information on field pictures and filtering, see Chapter 9, "Field Pictures and Filtering" and the description of the picture display-type option in Chapter 7, "Display Types." For information on range restrictions, see the description of the range display-type option in Chapter 7, "Display Types." Programmer-defined field validation routines are discussed later in this chapter.

If a field value fails one of the validation checks, the Forms Processor aborts the validation procedure, sounds the terminal bell, and redisplay the form. The cursor is positioned to the field that failed validation, and an error message is displayed at the bottom of the screen. The user can then either correct the field value and resubmit the form or cancel the form.

Programmer-Defined Field Validation Routines

You might want to establish restrictions on a field value that cannot be expressed using the range, cycle, and picture options. To enforce such restrictions, you can write a validation routine for the field.

A *validation routine* is a subroutine that examines a field value and determines if it is valid. You can specify a validation routine for a field in the **(MENU) F** form of the Forms Editor, or in the **validate** display-type option. If you establish a validation routine for a field, the Forms Processor automatically invokes that routine as the last step in the field validation suite.

A validation routine must have four parameters:

► **p_field_value_length** (input)

A two-byte integer indicating the length of the value specified for the field.

► **p_field_value_buffer** (input)

A 256-byte character string that holds the character-string representation of the field value.

► **p_message_length** (input-output)

A two-byte integer indicating the length of the error message returned to the Forms Processor. On input, this value is 0. If, on return to the Forms Processor, **p_message_length** is non-zero, this indicates that the field value is invalid.

► **p_message_buffer** (output)

An 80-byte character string to hold an error message.

On input, if the value of **p_field_value_length** is *N*, then the field value is stored in the first *N* bytes of **p_field_value_buffer**. On output, if the value of **p_message_length** is *M*, then the Forms Processor reads the error message from the first *M* bytes of **p_message_buffer**.

If **p_message_length** is non-zero when control returns to the Forms Processor, the Forms Processor stops field validation, sounds the terminal bell, and redisplay the form. The cursor is positioned to the field that failed validation, and the message in **p_message_buffer** is displayed at the bottom of the screen.

Note that the field value passed to the validation routine is always in unfiltered character-string format. For a numeric field, the validation routine can convert the value to an appropriate numeric data type.

Figure 10-1 shows a sample validation routine that checks if a two-byte integer field value is odd.

```
identification division.
program-id. enforce_oddness.

data division.
working-storage section.

01 field_contents      pic s9(5).
01 field_value         comp-4.

linkage section.

01 p_field_value_length comp-4.
01 p_field_value_buffer pic x(256) display.
01 p_message_length    comp-4.
01 p_message_buffer    pic x(80) display.

procedure division using p_field_value_length,
                        p_field_value_buffer, p_message_length,
                        p_message_buffer.

*   Move the numeric-string field value to field_contents.

    move p_field_value_buffer (:p_field_value_length)
      to field_contents.

*   Convert field_contents to a computational value and move
*   it to field_value.

    move !comp-4 (field_contents) to field_value.

*   If the field_value is even, report an error.

    if !mod (field_value, 2) equal 0 then
      move 'The value must be odd.' to p_message_buffer
      move !length('The value must be odd.') to
        p_message_length.

    exit program.
```

Figure 10-1. Sample Validation Routine

The sample in Figure 10-1 first moves the field value from `p_field_value_buffer` into a numeric display variable and then to a computational variable. The `mod` function is then used to determine if the value is odd or even.

Note: A validation routine should **not** depend on the results of the validation routine for another field because the order in which they are invoked is not defined. A validation routine **cannot** reference other information about the current form display, such as the key used to submit the form, or values given for other fields. If the validation restrictions for a field require such information, you must check the field value within the main program after control returns from the `perform screen input` statement.

Chapter 11:

Windows and Subforms

This chapter discusses the use of window fields to display subforms within a master form.

A *window* is a rectangular region of the screen in which an application can display a form. The entire screen is itself a window called the *master window*. A form displayed in this window is called a *master form*. A master form can contain a special kind of field called a *window field*. Such a field defines another window in which the application can display another form. A form displayed in a window within another form is called a *subform*. A subform can itself contain a window field in which another subform can be displayed.

Note: Any predefined form can be used as a master form or as a subform. These terms refer to how the form is used; they are not characteristics of the form itself. The window in which a form is displayed is determined when the form is initialized with the perform screen initialization statement.

A form can contain more than one window field.

The form that contains the window field in which a subform is displayed is the *immediately containing form* of that subform. The immediately containing form of a subform can be a master form or another subform.

Defining a Window Field

You can define a window field in a predefined form by using the `(MENU) I` request within the Forms Editor. Within the `(MENU) I` form, you must specify the location of the window field and the number of rows and columns it contains. The Forms Editor assigns a field ID to the window field so that you can reference it within a program.

For more information on the `(MENU) I` Forms Editor request, see Chapter 4, "The Forms Editor."

Within a program, you can create or modify a window field by using the `window` and `position` field options. The `window` option specifies the number of rows and columns in the window, and the `position` option specifies the location of the window.

The following field description defines a window field.

```
field (window_field_id) window (number_rows, number_columns)
    position (row_number, column_number)
```

Note that the `position` option specifies the position of the field within the **form**, rather than within the screen. The top left corner of the form is always row 1 and column 1.

For more information on the `window` and `position` field options, see Chapter 6, “Field Descriptions.”

Like any field, each row of a window field must be preceded and followed by a blank character position. This creates a column of blank positions to the left and right of the window. These columns are reserved for attribute bytes.

Recall that the first and last columns of a form are also reserved for attribute bytes. When you display a subform in a window, the Forms Processor overlays the first column of the subform onto the column of blank characters to the left of the window. If necessary, the Forms Processor also overlays the last column of the form onto the column of blank characters to the right of the window. Therefore, if a window field contains N columns, the widest subform that you can display in that window contains $N+2$ columns.

If a window contains M rows, the longest subform that you can display in that window also contains M rows.

Initializing the Forms

The initialization for a master form requires no special handling. You initialize it with the `perform screen initialization` statement as you would any ordinary form.

However, when you initialize a subform, you must indicate in what window the form will be displayed. The `origin` form option allows you to specify the form ID of the immediately containing form and the field ID of the appropriate window field.

```
perform screen initialization 'subform_name'
    with formid (subform_id)
        origin (containing_form_id, window_field_id)
        status (error_code).
```

You can omit the form ID of the containing form from the `origin` option if the field ID of the window field is unambiguous. However, it is a good practice to

always include the containing form ID. This makes the program easier to read and maintain.

For more information on the `origin` option, see Chapter 5, "Form Options."

You can initialize a subform any time after the immediately containing form is initialized.

Normally, no more than one form is active for each window. If you wish to maintain additional active forms, you must save forms explicitly. For more information, see Chapter 12, "Form Caching."

If you want to display a single form in two different windows within an application, you must initialize the form once for each window. The Forms Processor assigns a unique form ID for each initialization and treats the two occurrences of the form as two different forms.

Displaying the Forms

You can display a master form, as you would any ordinary form, using the `perform screen input` or `perform screen output` statement. When you display the master form, all window fields in the form are initially empty.

If you display the master form with the `perform screen input` statement, the user can modify any input fields in the form. The user cannot position to the window field.

You can display a subform only when its immediately containing form is on the screen. You can display the form with the `perform screen input` or `perform screen output` statement as you would any other form. The Forms Processor displays the form in the location specified in the `perform screen initialization` statement.

The user can modify fields in only one form at a time. If you display the subform with the `perform screen input` statement, the user can modify any input fields in the subform. The user cannot position to any fields in the containing form. If you want to allow the user to modify fields in the containing form, you must redisplay that form with the `perform screen input` statement. Fields in the subform are then inaccessible.

Although the user cannot modify fields in both a subform and its containing form on the same form display, you can often effectively overcome this restriction by using the `VERTICAL_SCROLL_TRAP` options switch. For information on this switch, see Chapter 13, "Traps."

Example of Windows

This section shows an example program that displays a master form with a single window. Depending on the user's action, the program then displays one of two subforms in the window.

The master form is called `employee_master`. It displays an employee's name and employee ID number, along with instructions to the user. A window field called `info_window` appears in the bottom part of the form. The form layout is as follows:

The diagram shows a rounded rectangular form. At the top, it has two input fields: "Employee name:" followed by a long horizontal line, and "Employee number:" followed by a short horizontal line. Below these are two text labels: "FUNCT-1: Display personal information" and "FUNCT-2: Display employment information". At the bottom, there is a large, empty rectangular box representing a window field.

In the example, the window field is represented by a rectangle. When the form is displayed on the screen, the window is invisible.

The first subform that can be displayed in the window is called `personal_info`. It displays the employee's home address and phone number. The layout of the `personal_info` form is as follows:

The diagram shows a rounded rectangular form. It contains four input fields: "Address:" followed by a long horizontal line, "City:" followed by a horizontal line, "State:" followed by a short horizontal line, and "Zip:" followed by a horizontal line with a hyphen. Below these is a "Phone:" label followed by a horizontal line with two hyphens.

The second subform is called `employment_info`. It displays the employee's department number, Social Security number, and salary. The layout of the `employment_info` form is as follows:

Department Number: _____ Social Security Number: ____ - ____ - ____

Salary: \$ _____ . ____ per month

The program displays `employee_master` and accepts input from the user. If the user submits the form with the first masked key, the program displays the `personal_info` form in the window. If the user submits the master form with the second masked key, the program displays the `employment_info` form in the window. After displaying the appropriate subform, the program loops back and redisplay the master form to allow the user to make another selection.

The program is shown in Figure 11-1.

```
identification division.
program-id. display_employee_info.

data division.
working-storage section.

        %replace CANCEL                by -1
        %replace ENTER                by 0

        copy 'employee_master_ids'.

01 employment_fields.
   copy 'employee_info'.
01 master_fields.
   copy 'employee_master'.
01 personal_fields.
   copy 'personal_info'.

01 error_code          comp-4.
01 key_code            comp-4.
01 master_form_id      comp-4.
01 subform_id          comp-4.

procedure division.

*   Initialize the master form.

        perform screen initialization 'employee_master'
               into (master_fields)
               with formid (master_form_id) status (error_code).
```

Figure 11-1. Example Program Using Subforms

(Continued on next page)

Figure 11-1. (Continued)

```

        if error_code not equal 0 then
            go to fatal-error.

        move ENTER to key_code.
        move 0 to error_code.

    *   Display the master form and any requested subforms.

        perform input-loop until ((key_code equal CANCEL) or
                                   (error_code not equal 0)).
        if error_code not equal 0 then
            go to fatal-error.

        go to exit-program.

input-loop.

        perform screen input 'employee_master'
            update (master_fields)
            with keyused (key_code) status (error_code).

        if error_code not equal 0 then
            go to fatal-error.

    *   Examine the value of key_code to determine user's choice.
    *   Initialize and display the appropriate subform.

        if key_code equal 1 then
            perform personal-display
        else if key_code equal 2 then
            perform employment-display.

personal-display.

    *   Initialize the personal_info form in info_window
    *   within the employee_master form.

        perform screen initialization 'personal_info'
            with origin (master_form_id, info_window_id)
            formid (subform_id) status (error_code).

```

(Continued on next page)

Figure 11-1. (Continued)

```
        if error_code not equal 0 then
            go to fatal-error.

*   Set the members of personal_fields to the appropriate values.
    .
    .
    .

*   Display the personal_info form.

    perform screen output 'personal_info'
        update (personal_fields)
        with formid (subform_id) status (error_code).

    if error_code not equal 0 then
        go to fatal-error.

employment-display.

*   Initialize the employment_info form in info_window
*   within the employee_master form.

    perform screen initialization 'employment_info'
        with origin (master_form_id, info_window_id)
        formid (subform_id) status (error_code).

    if error_code not equal 0 then
        go to fatal-error.

*   Set the members of employment_fields to the appropriate values.
    .
    .
    .

*   Display the employment_info form.

    perform screen output 'employment_info'
        update (employment_fields)
        with formid (subform_id) status (error_code).
```

(Continued on next page)

Figure 11-1. (Continued)

```
        if error_code not equal 0 then
            go to fatal-error.

        fatal-error.

        *   Handle error.
            .
            .
            .

        exit-program.

        exit program.
```

If you expect the user to switch back and forth between the two subforms, it is more efficient to initialize and cache both subforms before entering the main loop. This allows you to remove the perform screen initialization statements from inside the loop. See Figure 12-1 in Chapter 12, "Form Caching."

Chapter 12:

Form Caching

This chapter describes how to save forms that are not currently in use so that they can be used later without re-initializing.

At any time, an application can have several active forms. A display list for each active form is allocated in the user heap. For each window, the Forms Processor maintains one current form. The *current form* of a window is the form most recently displayed or initialized in that window.

A form first becomes active when it is initialized with the `perform screen initialization` statement. At that time, the form also becomes the current form for the window in which it is initialized.

Normally, a form remains active only when it is the current form for a window. If a second form becomes the current form for the window, the Forms Processor discards the first form. This means that if you want to reference the first form again, you must re-initialize it. Re-initialization consumes system time and resources.

You can prevent the Forms Processor from discarding a form by explicitly *caching*, or saving, the form. A cached form remains active even when it is not the current form for the window. You can later display the cached form without re-initializing it. When you display a cached form, that form becomes the current form for the window.

To cache a form, use the `perform screen save` statement as follows:

```
perform screen save [formid (form_id) ] [portid (port_id) ]  
                    [status (status_code) ] .
```

If you omit the `formid` option of the `perform screen save` statement, the most recently referenced form is cached. However, in any application that uses more than one form, you should include the `formid` option in all screen statements to help avoid errors and to make the program easier to read and maintain.

You can discard a cached form by issuing the `perform screen discard` statement as follows:

```
perform screen discard [formid (form_id) ] [portid (port_id) ]  
                        [status (status_code) ] .
```

Discarding a form frees heap space. If an application displays many forms, you might have to discard forms that are no longer needed to conserve space.

The Forms Processor determines whether to discard a form by examining an internal reference count associated with the form. This reference count is incremented when a form becomes the current form for a window and is decremented when the form is no longer current. When the reference count becomes less than or equal to 0, the form is discarded. The `perform screen save` statement increments this reference count by 1, and the `perform screen discard` statement decrements the count by 1. Therefore, one `perform screen discard` statement cancels the effect of exactly one `perform screen save` statement.

The following section describes in detail how the Forms Processor manipulates the reference count.

The Forms Reference Count

You cannot read or directly address the reference count for a form. It is read and updated only by the Forms Processor.

When a form becomes the current form for a window, its reference count is incremented. When a form is replaced as the current form, its reference count is decremented. For a master form, the amount of increment or decrement is always 1. For a subform, the amount of increment or decrement is the value of the reference count of the immediately containing form.

The `perform screen save` statement increments a form's reference count by 1. The `perform screen discard` statement decrements a form's reference count by 1.

If a form contains one or more window fields, then whenever the reference count of that form changes, the reference count of the current form in each window of that form also changes by the same amount. This rule is applied recursively to nested subforms.

Whenever the reference count of any form becomes less than or equal to 0, the form is discarded.

When program execution ends, all active forms are discarded.

Consider an application that displays the following forms: main1, main2, sub1, sub1a, and sub2. Table 12-1 shows the effect of specific actions on the form reference counts. In the figure, a hyphen represents the reference count for an inactive form.

Table 12-1. Form Reference Count Manipulation

Action	Reference Counts				
	main1	sub1	sub1a	sub2	main2
Initialize main1 in master window	1	—	—	—	—
Initialize sub1 in window within main1	1	1	—	—	—
Save sub1	1	2	—	—	—
Initialize sub2 in same window as sub1	1	1	—	1	—
Save sub2	1	1	—	2	—
Display main1	1	1	—	2	—
Display sub1 within main1	1	2	—	1	—
Save main1	2	3	—	1	—
Initialize sub1a in window within sub1	2	3	3	1	—
Display sub1a within sub1	2	3	3	1	—
Display sub2 in place of sub1	2	1	1	3	—
Initialize main2 in master window	1	1	1	2	1
Display main2	1	1	1	2	1
Discard main1	—	1	1	1	1
Discard sub1	—	—	—	1	1
Discard sub2	—	—	—	—	1
Discard main2	—	—	—	—	—

Example of Form Caching

The example program at the end of Chapter 11, “Windows and Subforms” displays a master form with a window field in which either of two subforms can be displayed. The program contains a loop so that the user can switch back and forth between the two subforms. In the example, the chosen subform is initialized on each pass through the loop. A better strategy is to initialize and cache both subforms before entering the loop. This means that either subform can be displayed within the loop without re-initializing.

Figure 12-1 shows an updated version of the example program using form caching.

```
identification division.
program-id. display_employee_info.

data division.
working-storage section.

        %replace CANCEL                by -1
        %replace ENTER                 by 0

        copy 'employee_master_ids'.

01 employment_fields.
   copy 'employee_info'.
01 master_fields.
   copy 'employee_master'.
01 personal_fields.
   copy 'personal_info'.

01 error_code          comp-4.
01 key_code            comp-4.
01 master_form_id      comp-4.
01 employment_form_id  comp-4.
01 personal_form_id    comp-4.

procedure division.

*   Initialize the master form.

        perform screen initialization 'employee_master'
                into (master_fields)
                with formid (master_form_id) status (error_code).

        if error_code not equal 0 then
                go to fatal-error.
```

Figure 12-1. Example Program Using Subforms

(Continued on next page)

Figure 12-1. (Continued)

```
*   Initialize the subforms with the master form and save the subforms.

perform screen initialization 'personal_info'
    with origin (master_form_id, info_window_id)
    formid (personal_form_id) status (error_code).

if error_code not equal 0 then
    go to fatal-error.

perform screen save with formid (personal_form_id)
    status (error_code).

if error_code not equal 0 then
    go to fatal-error.

perform screen initialization 'employment_info'
    with origin (master_form_id, info_window_id)
    formid (employment_form_id) status (error_code).

if error_code not equal 0 then
    go to fatal-error.

perform screen save with formid (employment_form_id)
    status (error_code).

if error_code not equal 0 then
    go to fatal-error.

move ENTER to key_code.
move 0 to error_code.
```

(Continued on next page)

Figure 12-1. (Continued)

```
*   Display the master form and any requested subforms.

    perform input-loop until ((key_code equal CANCEL) or
                             (error_code not equal 0)).
    if error_code not equal 0 then
        go to fatal-error.

    go to exit-program.

input-loop.

    perform screen input 'employee_master'
        update (master_fields)
        with keyused (key_code) status (error_code).

    if error_code not equal 0 then
        go to fatal-error.

*   Examine the value of key_code to determine user's choice.
*   Initialize and display the appropriate subform.

    if key_code equal 1 then
        perform personal-display
    else if key_code equal 2 then
        perform employment-display.

personal-display.

*   Set the members of personal_fields to the appropriate values.
    .
    .
    .

*   Display the personal_info form.

    perform screen output 'personal_info'
        update (personal_fields)
        with formid (personal_form_id) status (error_code).

    if error_code not equal 0 then
        go to fatal-error.
```

(Continued on next page)

Figure 12-1. (Continued)

```
employment-display.  
  
*   Set the members of employment_fields to the appropriate values.  
    .  
    .  
    .  
  
*   Display the employment_info form.  
  
    perform screen output 'employment_info'  
        update (employment_fields)  
        with formid (employment_form_id) status (error_code).  
  
    if error_code not equal 0 then  
        go to fatal-error.  
  
fatal-error.  
  
*   Handle error.  
    .  
    .  
    .  
  
    go to exit-program.  
  
exit-program..  
  
exit program.
```

Note that you do not need to cache the master form in the example program shown in Figure 12-1, because no other form is initialized in the master window.

)

)

)

)

)

Chapter 13:

Traps

In FMS, a *trap* is a return of control from a form to the application program before the user submits or cancels the form. You can establish three kinds of traps in a form.

- Trap on field entry. A trap that occurs each time the user positions to a specific field.
- Trap on field exit. A trap that occurs each time the user positions out of a specific field.
- Vertical scroll trap. A trap that occurs each time the user tries to position the cursor to before the first line of the form, or beyond the last line of the form.

Typically, when a trap occurs, the program updates the display list and then executes another `perform screen input` statement to redisplay the form. Traps can allow the program to update a partially completed form based on the information the user has filled in up to that point.

Note: Some device drivers do not handle traps.

Establishing Traps

Traps on field exit and field entry are controlled by display-type action switches. You can establish a trap on field exit by setting the `TRAP_ON_FIELD_EXIT` action switch for the field to true. Likewise, you can establish a trap on field entry by setting the `TRAP_ON_FIELD_ENTRY` action switch for the field to true. For information on these switches, see Chapter 7, “Display Types.”

For a predefined field, you can specify an initial value for the `TRAP_ON_FIELD_EXIT` and `TRAP_ON_FIELD_ENTRY` switches with the `TRAP ON FIELD EXIT` and `TRAP ON FIELD ENTRY` options of the Forms Editor `(MENU) F` form. For information on these options, see Chapter 4, “The Forms Editor.”

The field for which a trap on field entry or trap on field exit is set is called the *trap field*.

The vertical scroll trap is controlled by the `VERTICAL_SCROLL_TRAP` switch in the options form option. See the description of the options form option in Chapter 5, “Form Options.”

For a predefined form, you can specify an initial value for the `VERTICAL_SCROLL_TRAP` switch with the `VERTICAL_SCROLL_TRAP` option of the Forms Editor `(MENU)`s form. For information on this option, see Chapter 4, “The Forms Editor.”

Trap on Field Entry

If the `TRAP_ON_FIELD_ENTRY` action switch is true for a field in a displayed form, then a trap occurs when the user attempts to move the cursor into that field from another field. If, on return to the form from the trap, the cursor is positioned to the trap field, the trap does not recur. The trap is not activated again unless the user positions the cursor out of the trap field and then back into it.

When a trap on field entry occurs, control returns to the program. The current field values are returned in the update option of the perform screen input statement that displayed the form.

If the perform screen input statement contains the `keyused` form option, the value `-9` is returned in that option.

If the perform screen input statement contains the `nextcursor` form option, the value returned in that option is the field ID of the trap field. This is typically the field to which you should initially position the cursor on a subsequent display of the form.

If the perform screen input statement contains the `getcursor` option, the value returned in that option is also the field ID of the trap field.

If the perform screen input statement contains the `functionkey` form option, the value returned in that option indicates what generic sequence the user issued to move the cursor into the field.

Note: If the `TRAP_ON_FIELD_ENTRY` switch is true for the field to which the cursor is initially positioned in a perform screen input statement, then the trap is activated immediately: the Forms Processor returns control to the program without displaying the form. If the cursor is positioned to that same field for the next perform screen input statement, then the trap is not activated, and the form is displayed.

Trap on Field Exit

If the `TRAP_ON_FIELD_EXIT` action switch is true for a field in a displayed form, then a trap occurs when the user attempts to move the cursor from that field to another field. A trap also occurs if the Forms Processor attempts to move the cursor to another field as the result of the auto-tab feature. For information on auto-tab, see the description of the `AUTO_TAB_TO_NEXT_FIELD` action switch in Chapter 7, “Display Types.”

Returned Values

When a trap on field exit occurs, control returns to the program. The current field values are returned in the update option of the perform screen input statement that displayed the form.

If the perform screen input statement contains the `keyused` form option, the value -2 is returned in that option.

If the perform screen input statement contains the `nextcursor` form option, the value returned in that option is the field ID of the field to which the cursor would have moved if no trap had occurred. This is typically the field to which you should initially position the cursor on a subsequent display of the form.

If the perform screen input statement contains the `getcursor` option, the value returned is the field ID of the trap field.

If the perform screen input statement contains the `functionkey` form option, the value returned in that option indicates what generic sequence the user issued to move the cursor out of the field.

Field Validation

When a trap on field exit occurs, the Forms Processor can validate all fields in the form or it can validate only the trap field. The `VALIDATE_ERRORS_OFF` and `VALIDATE_ONE_FIELD` switches of the options form option determine the level of validation.

The validation levels are as follows:

- Validate all fields in the form and report any validation errors (the default; both `VALIDATE_ONE_FIELD` and `VALIDATE_ERRORS_OFF` are false).
- Validate only the trap field and report any validation error (`VALIDATE_ONE_FIELD` is true; `VALIDATE_ERRORS_OFF` is false).
- Execute the validation routines for all fields in the form, but report errors only for the trap field (`VALIDATE_ERRORS_OFF` is true; `VALIDATE_ONE_FIELD` is false).

You cannot set both the `VALIDATE_ERRORS_OFF` switch and the `VALIDATE_ONE_FIELD` switch to true.

If you disable the reporting of validation errors for other fields by setting either `VALIDATE_ONE_FIELD` or `VALIDATE_ERRORS_OFF` to true, no errors are returned for any of the validation checks on those fields. This means that a required field can contain a null value, a field can have a value outside its prescribed range or inconsistent with its picture, or a field can fail the checks in a validation routine that you have written.

Note that the only significant difference between `VALIDATE_ERRORS_OFF` and `VALIDATE_ONE_FIELD` is whether field validation routines that you have written are invoked. You might choose to set `VALIDATE_ERRORS_OFF`, rather than `VALIDATE_ONE_FIELD`, if some of the field validation routines have side effects that you want to preserve.

You can initialize the values of the `VALIDATE_ERRORS_OFF` and `VALIDATE_ONE_FIELD` switches with the `VALIDATE` option in the `(MENU)`s form of the Forms Editor. For information on this option, see Chapter 4, “The Forms Editor.”

For information on the options form option, see Chapter 5, “Form Options.”
For information on field validation, see Chapter 10, “Error Handling and Field Validation.”

Example Using a Trap on Field Exit

The sample application developed in Chapter 2, “Developing an FMS Application,” uses a form that allows the user to enter information about an employee. One piece of information in the form is the employee’s department number. As an aid to the user, you might want to display the name of the department after the user has selected the department number. To do this, modify the `employee_info` form as follows:

- Add a new output field named `department_name`.
- Modify the `department_number` field, setting the `TRAP_ON_FIELD_EXIT` switch to true.
- Set the `VALIDATE_ONE_FIELD` options switch to true for the form.

Within the application program, add the `nextcursor` form option to the `perform screen input` statement. Add a new case to the `if` statement to handle a trap on field exit. When the trap occurs, examine the value of `department_number` of `employee_fields` to determine what value the user chose for the `department_number` field. Obtain the corresponding department name, and assign the name to `department_number` of `employee_fields`. Assign the field ID returned in the `nextcursor` option to `cursor_field`, and loop back to the `perform screen input` statement to redisplay the form. The department name now appears on the user’s screen, and the cursor is positioned as the user would expect.

Vertical Scroll Trap

If the `VERTICAL_SCROLL_TRAP` options form option switch is true for a form display, then a trap occurs if the user attempts to position the cursor before the first field in the form or after the last field in the form.

If the `VERTICAL_SCROLL_TRAP` switch is false, then if the user tries to position before the first field in the form, the cursor moves to the bottom of the form. If the user tries to position after the last field in the form, the cursor moves to the top of the form.

The Forms Processor treats a vertical scroll trap exactly the same as it treats a trap on field exit. The field from which the cursor is being moved is treated as the trap field. The field ID of this field is returned in the `getcurs` form option.

See the “Trap on Field Exit” section earlier in this chapter for information on the values returned to the program and on the level of field validation performed for a trap on field exit.

Scrolling between Forms

You can use the `VERTICAL_SCROLL_TRAP` switch to allow the user to scroll between two forms in a single window. This creates the appearance of a single form that is longer than the window. You can also use the `VERTICAL_SCROLL_TRAP` switch to allow the user to move freely between two forms in different windows, such as a master form and a subform.

To create the appearance of a form that is longer than the window in which it is displayed, first plan the layout of the long form. Next, divide the long form into two or more forms that each fit in the window that is available for the form. Define each of these small forms with the `VERTICAL_SCROLL_TRAP` switch set to true. Design the application to initially display the topmost of the small forms. Whenever the user attempts to move beyond the bottom of a form, display the next small form. If the user attempts to move beyond the top of a small form, display the previous small form.

For example, suppose the form you plan is twice as long as the screen. Divide the long form into two full-screen forms, top and bottom. When you define the forms, set the `VERTICAL_SCROLL_TRAP` switch to true for each. Within the application program, initially display top. If a trap occurs because the user attempted to move beyond the end of top, display bottom with the cursor positioned to the first field. If a trap subsequently occurs because the user attempted to move beyond the beginning of bottom, display top with the cursor positioned to the last field. To the user, top and bottom appear to be one continuous form.

Note: When the user submits a form, the Forms Processor validates only fields in that particular form. If fields in the other form or forms also require validation, the program must perform that validation itself and handle errors appropriately. You should also consider what type of validation the Forms Processor should perform when you scroll from form to form. See the “Field Validation” subsection under “Trap on Field Exit” earlier in this chapter for more information.

If you are displaying a master form and a subform on the screen, although both forms are visible on the screen, the user can normally modify only one of the forms at a time. The user must submit one form before moving into the other. By using the `VERTICAL_SCROLL_TRAP` switch for both forms, you can effectively allow the user to move freely from one form to the other. For example, assume the subform is in a window at the bottom of the master form. If the user attempts to move beyond the end of the master form, display the subform with the cursor in the first field. If the user attempts to move beyond the beginning of the master form, display the subform with the cursor in the last field. If the user attempts to move beyond the end of the subform, display the master form with the cursor in the first field. Finally, if the user attempts to move beyond the beginning of the subform, display the master form with the cursor in the last field before the window.

Note that the same field validation considerations apply as when scrolling between two forms in the same window.

Scrolling within a Form

You can also use the `VERTICAL_SCROLL_TRAP` switch to scroll a list of values through a single form. For example, you could define a form, `string_list`, that contains a vertical array field in which you can display a list of values (one value in each array element). The user can select one of these values by positioning the cursor to it and submitting the form. However, assume you have a list of 50 values to be displayed, and the array field has only 20 elements. By using the `VERTICAL_SCROLL_TRAP` switch, you can allow the user to scroll through all 50 values.

Initially, display the first 20 values in the array field. If the user selects one of these values and submits the form, then no scrolling is necessary. However, if the user attempts to position beyond the last field, a trap occurs. At this point, the program can load the next 20 values into the array field and redisplay the form with the cursor at the top of the screen. If the user subsequently attempts to move beyond the beginning of the form, display the first 20 values again. If the user attempts to

move beyond the end of the second screen of values, display the final 10 values. Optionally, you might choose to create overlap between the last two screens by displaying the last 20 values on the last screen, rather than leaving 10 fields blank.

Figure 13-1 illustrates this algorithm. The form `string_list` contains an array field named `strings`. The values to be displayed in `strings` are stored in the table `values`. In the `perform` screen input statement, the code returned in the `keyused` option indicates whether a vertical scroll trap has occurred, and the field ID returned in `nextcursor` indicates the direction in which to scroll.

```

identification division.
program-id. vscroll.

data division.
working-storage section.

        %replace CANCEL          by -1
        %replace SCROLL_TRAP    by -2

        copy 'form_options.incl.cobol'.
        copy 'string_list_ids.incl.cobol'.

01 fields.
   copy 'string_list'.

01 bottom_value          comp-4.
01 top_value             comp-4.

01 values_table.
   02 value_string       pic x(64) display-2 occurs 50 times.

01 error_code            comp-4.
01 i                     comp-4.
01 key_code              comp-4.
01 next_cursor           comp-4.

```

Figure 13-1. Vertical Scroll Trap Example Program

(Continued on next page)

Figure 13-1. *(Continued)*

```
procedure division.  
  
*   Initialize values_table.  
    .  
    .  
    .  
  
*   Initialize the display list.  
  
    perform screen initialization 'string_list'  
        with status = error_code.  
  
    if error_code not equal 0 then  
        go to fatal-error.  
  
    move 1 to next_cursor. /* cursor position for next form display */  
    move 1 to key_code.    /* code returned by most recent display */  
  
*   The variables top_value and bottom_value are indexes into  
*   the values_table. Set top_value to the index of the first  
*   value on the screen and bottom_value to the index of the  
*   last value on the screen.  
  
    move 1 to top_value.  
    move strings_ct to bottom_value.  
  
*   Assign beginning of value list to strings of fields.  
  
    perform assign-field-value varying i from 1  
        until i is greater than strings_ct.
```

(Continued on next page)

Figure 13-1. (Continued)

```

*   Display the form until user cancels or an error occurs.

      perform input-loop until ((key_code equal CANCEL) or
                               (error_code not equal 0)).

      if error_code not equal 0 then
        go to fatal-error.

      go to exit-program.

assign-field-value.

      move value_string (top_value - 1 + i) to strings_1 (i).

input-loop.

      perform screen input 'string_list' update (fields)
        with keyused (key_code) nextcursor (next_cursor)
        putcursor (next_cursor) status (error_code)
        options (VERTICAL_SCROLL_TRAP, WIDE_CURSOR).

      if error_code not equal 0 then
        go to fatal-error.

      if key_code equal SCROLL_TRAP then
        if next_cursor equal 1 then
          perform scroll-down
        else perform scroll-up
      else /* Handle other key_code values. */.

scroll-down.

      if bottom_value is less than 50 then
        compute bottom_value =
          !min (50, bottom_value + strings_ct)
        compute top_value = bottom_value - strings_ct + 1.

      perform assign-field-value varying i from 1
        until i is greater than strings_ct
      else compute next_cursor =
        strings_id + strings_ct - 1. /* bottom of form */

```

(Continued on next page)

Figure 13-1. (Continued)

```

scroll-up.

    if top_value is greater than 1 then
        compute top_value = !max(1, top_value - strings_ct)
        compute bottom_value = top_value + strings_ct - 1.

        perform assign-field-value varying i from 1
            until i is greater than strings_ct
        else move strings_id to next_cursor. /* top of form */

fatal-error.

*   Handle error.
    .
    .
    .

exit-program.

exit program.

```

The example in Figure 13-1 is somewhat simplified. In particular, the cursor positioning is not always as expected.

You might want to change the behavior that occurs when the user tries to position before the first screen or beyond the last screen. You can create a circular list by displaying the last screen when the user attempts to position before the first screen, and displaying the first screen when the user attempts to position beyond the last screen.

Forms Input Mode

When using a trap in a form, you should carefully consider the use of forms input mode. Forms input mode can be convenient because it allows the user to type ahead while the program is processing the trap. However, if the processing of the trap might change the form in a way that the user would not expect, allowing this ability to type ahead could lead to user errors.

You must decide for each application whether forms input mode is helpful. For example, in the sample program shown in Figure 13-1, forms input mode would allow the user to scroll through the list of values faster.

For information on forms input mode, see Chapter 14, “Subroutines.”

Chapter 14:

Subroutines

This chapter describes two subroutines that change the channel input mode for a forms application. Changing the input mode affects the way VOS handles characters that are typed between form displays.

Forms Input Mode

Many FMS applications display more than one form or redisplay the same form several times. Normally, if the user types some characters after submitting one form and before the Forms Processor displays the next form, those characters are not read by the Forms Processor. The characters are echoed to the screen, possibly corrupting the subsequent form display.

However, if the communications channel is in *forms input mode*, the characters typed after the form is submitted are not echoed to the screen. Instead, they are saved until the next form is displayed. They are then read by the Forms Processor and applied to the form. This allows the user to anticipate the next form and begin entering input before the form is displayed.

To put the channel associated with your terminal port into forms input mode, call the subroutine `s$begin_forms_input`. To take the channel out of forms input mode, call `s$end_forms_input`. You can change any other channel to or from forms input mode by calling the subroutine `s$control` with the `SET_MODES_OPCODE` (207) or the `SET_INFO_OPCODE` (202). For information on `s$control`, see the *VOS Communications Software: Asynchronous Communications (R025)*.

While the channel is in forms input mode, all input for the channel must be done through forms. If you attempt to perform non-forms input (for example, by calling `s$read`), that input will fail. Forms input mode does not restrict output operations.

When the channel first enters or exits forms input mode, any currently pending forms input is discarded. The Forms Processor also discards pending input if the beep option is true for a form display. You can use the beep option when an error or other exceptional condition changes the usual sequence of form displays.

Forms input mode is sometimes called *continuous forms mode*.

Note: Forms input mode is not applicable to some device drivers.

s\$begin_forms_input

Purpose

The s\$begin_forms_input subroutine puts the channel associated with your terminal port into forms input mode.

Usage

01 error_code	comp-4.
call 's\$begin_forms_input' using error_code.	

Arguments

► error_code (output)

A returned status code.

Explanation

The s\$begin_forms_input subroutine puts the channel associated with the terminal port of the current process into forms input mode.

If the channel is already in forms input mode, a call to s\$begin_forms_input has no effect and returns the value 0 in error_code.

Error Codes

The s\$begin_forms_input subroutine returns standard VOS error codes.

s\$end_forms_input

s\$end_forms_input

Purpose

The s\$end_forms_input subroutine removes the channel associated with your terminal port from forms input mode.

Usage

01	error_code	comp-4.
	call 's\$end_forms_input' using error_code.	

Arguments

► error_code (output)

A returned status code.

Explanation

The s\$end_forms_input subroutine removes the channel associated with the terminal port of the current process from forms input mode.

If the terminal port is not in forms input mode when you call s\$end_forms_input, the subroutine has no effect and returns the value 0 in error_code.

Error Codes

The s\$end_forms_input subroutine returns standard VOS error codes.

Chapter 15:

Built-In Functions

Fourteen built-in functions have been added to VOS COBOL to simplify the handling of fields and display types in FMS applications.

Conceptually, the Forms Processor maintains three lists that you can reference through built-in functions.

- A list of all global display types that are currently allocated. This list includes both reserved global display types and programmer-defined display types. It is ordered numerically.
- A list of all fields currently in the form. This list includes programmer-defined fields, and fields defined by the Forms Processor for background text. The fields are sorted in display order.
- A list of all fields that were changed during the most recent display of the form. The order of this list is undefined.

You can use built-in functions to step through all the values in each of these lists. You can also add to the first two lists by using built-in functions to allocate new fields and new programmer-defined display types.

Other built-in functions allow you to find the field ID for a named field or to find the field in a specific position.

Built-In Function Summary

The display-type built-in functions allow you to do the following:

- find the next available unused global display-type ID and mark it as in-use
- find all the global display-type IDs that are currently in use.

The field built-in functions allow you to do the following:

- find the next available unused field ID for a form and mark it as in-use
- find all the field IDs that are currently in use for a form
- find the field IDs of all fields that were changed in the most recent form display
- find the field ID of a field with a specific name
- find the field ID of an input field in a specific location within the form.

Table 15-1 and Table 15-2 list the display-type built-in functions and the field built-in functions, respectively.

Table 15-1. Display-Type Built-In Functions

Function	Explanation
<code>alloc_screen_displaytype</code> ([<i>port_id</i>])	Allocates a new display type, and returns the display-type ID.
<code>first_screen_displaytype</code> ([<i>port_id</i>])	Returns the lowest display-type ID currently allocated.
<code>last_screen_displaytype</code> ([<i>port_id</i>])	Returns the highest display-type ID currently allocated.
<code>next_screen_displaytype</code> (<i>display_type_id</i> [, <i>port_id</i>])	Returns the next highest display-type ID currently allocated.
<code>prev_screen_displaytype</code> (<i>display_type_id</i> [, <i>port_id</i>])	Returns the next lowest display-type ID currently allocated.

Table 15-2. Field Built-In Functions

Function	Explanation
<code>alloc_screen_field</code> ([<i>form_id</i> [, <i>port_id</i>]])	Allocates a new field and returns the field ID.
<code>find_screen_field</code> (<i>field_name</i> [, <i>form_id</i> [, <i>port_id</i>]])	Returns the field ID for the named field.
<code>first_changed_field</code> ([<i>form_id</i> [, <i>port_id</i>]])	Returns the lowest field ID of fields that have changed.
<code>first_screen_field</code> ([<i>form_id</i> [, <i>port_id</i>]])	Returns the lowest field ID currently allocated.
<code>last_screen_field</code> ([<i>form_id</i> [, <i>port_id</i>]])	Returns the highest field ID currently allocated.
<code>next_changed_field</code> (<i>field_id</i> [, <i>form_id</i> [, <i>port_id</i>]])	Returns the next highest field ID of fields that have changed.
<code>next_screen_field</code> (<i>field_id</i> [, <i>form_id</i> [, <i>port_id</i>]])	Returns the next highest field ID currently allocated.
<code>prev_screen_field</code> (<i>field_id</i> [, <i>form_id</i> [, <i>port_id</i>]])	Returns the next lowest field ID currently allocated.
<code>screen_field_position</code> (<i>row</i> , <i>column</i> [, <i>form_id</i> [, <i>port_id</i>]])	Returns the field ID of the first input field located at or before the specified position.

Built-In Function Reference Guide

This section describes each of the VOS COBOL FMS built-in functions. The functions are described in alphabetical order.

`alloc_screen_displaytype ([port_id])`

This function returns the lowest currently unused global display-type ID for the most recently referenced form and marks that display-type ID as being in use.

The value of *port_id*, if given, must be the integer ID of the port on which the form is to be displayed. If *port_id* is omitted or is less than 0, the default is the ID of the port most recently operated on by a screen or accept statement.

Display types are freed by the perform screen delete statement.

If a program allocates a predetermined number of display types, you do not need to use the `alloc_screen_displaytype` function. You can assign sequential display-type IDs (starting with 11) and use them in the `displaytype` option of an accept or screen statement to define the display type. Use the `alloc_screen_displaytype` function when you cannot determine the number of display types to be allocated until runtime.

For further information on display types, see Chapter 7, “Display Types.”

`alloc_screen_field ([form_id [, port_id]])`

This function returns the lowest currently unused field ID for the form and marks that field ID as being in use.

The value of *form_id*, if given, must be a valid form ID as returned by the perform screen initialization statement. If *form_id* is omitted or is less than 0, the ID of the most recently referenced form is used. The value of *port_id*, if given, must be the integer ID of the port on which the form is to be displayed. If *port_id* is omitted or is less than 0, the default is the ID of the port most recently operated on by a screen or accept statement.

Fields are freed by the perform screen delete statement.

Note: The Forms Processor uses some field IDs for form background text. Therefore, the field IDs returned by `alloc_screen_field` usually do not sequentially follow the field IDs assigned by the Forms Editor. You should **always** use the `alloc_screen_field` function when adding fields within the application program.

For further information on creating fields, see Chapter 6, “Field Descriptions.”

```
find_screen_field (field_name [ ,form_id [ ,port_id ] ] )
```

This function returns the field ID for the named field.

The value of *field_name* must be a field name given in the Forms Editor. The value of *form_id*, if given, must be a valid form ID as returned by the perform screen initialization statement. The value of *port_id*, if given, must be the integer ID of the port on which the form is to be displayed. If *port_id* is omitted or is less than 0, the default is the ID of the port most recently operated on by a screen or accept statement.

If the named field is not found for any reason, the function signals the error condition.

For information on the Forms Editor, see Chapter 4, "The Forms Editor."

```
first_changed_field ( [ form_id [ ,port_id ] ] )
```

This function returns the field ID of the first field in the internal changed-field list for the form. The returned value can be used as a seed or initial input value for the *next_changed_field* function.

The value of *form_id*, if given, must be a valid form ID as returned by the perform screen initialization statement. If *form_id* is omitted or is less than 0, the ID of the most recently referenced form is used. The value of *port_id*, if given, must be the integer ID of the port on which the form is to be displayed. If *port_id* is omitted or is less than 0, the default is the ID of the port most recently operated on by a screen or accept statement.

The internal changed-field list contains all of the fields that were changed in the most recent display of the form. The order of the list is undefined. If there are no changed fields, the *first_changed_field* function returns the null field ID (-32768).

You can retrieve the IDs of all changed fields in the form by invoking *first_changed_field*, followed by a series of invocations of *next_changed_field*. See the example in the description of the *next_changed_field* function later in this section.

`first_screen_displaytype ([port_id])`

This function returns the lowest display-type ID from the list of global display types. The returned value can be used as a seed or initial input value for the `next_screen_displaytype` function.

The value of *port_id*, if given, must be the integer ID of the port on which the form is to be displayed. If *port_id* is omitted or is less than 0, the default is the ID of the port most recently operated on by a screen or accept statement.

The list of global display types is sorted numerically.

If no global display types are currently allocated, `first_screen_displaytype` returns the null display-type ID (-32768).

You can retrieve all global display-type IDs currently in use (in numerical order) by invoking `first_screen_displaytype`, followed by a series of invocations of `next_screen_displaytype`. See the example in the description of the `next_screen_displaytype` function later in this section.

`first_screen_field ([form_id [, port_id]])`

This function returns the field ID of the first field in the display list. The returned value can be used as a seed or initial input value for calls to the `next_screen_field` function.

The value of *form_id*, if given, must be a valid form ID as returned by the `perform screen initialization` statement. If *form_id* is omitted or is less than 0, the ID of the most recently referenced form is used. The value of *port_id*, if given, must be the integer ID of the port on which the form is to be displayed. If *port_id* is omitted or is less than 0, the default is the ID of the port most recently operated on by a screen or accept statement.

The display list includes all fields the program can manipulate, plus all fields created by the Forms Processor for background text.

If there are no fields, the `first_screen_field` function returns the null field ID (-32768).

You can retrieve the IDs of all fields of a form (in display order) by invoking `first_screen_field`, followed by a series of invocations of `next_screen_field`. See the example in the description of the `next_screen_field` function later in this section.

`last_screen_displaytype ([port_id])`

This function returns the highest display-type ID from the list of global display types. The returned value can be used as a seed or initial input value for the `prev_screen_displaytype` function.

The value of *port_id*, if given, must be the integer ID of the port on which the form is to be displayed. If *port_id* is omitted or is less than 0, the default is the ID of the port most recently operated on by a screen or accept statement.

The list of global display types is sorted numerically.

If no global display types are currently allocated, `last_screen_displaytype` returns the null display-type ID (-32768).

You can retrieve all global display-type IDs currently in use (in reverse numerical order) by invoking `last_screen_displaytype`, followed by a series of invocations of `prev_screen_displaytype`. See the example in the description of the `prev_screen_displaytype` function later in this section.

`last_screen_field ([form_id [, port_id]])`

This function returns the last field ID in the display list. The returned value can be used as a seed or initial input value for the `prev_screen_field` function.

The value of *form_id*, if given, must be a valid form ID as returned by the `perform screen initialization` statement. If *form_id* is omitted or is less than 0, the ID of the most recently referenced form is used. The value of *port_id*, if given, must be the integer ID of the port on which the form is to be displayed. If *port_id* is omitted or is less than 0, the default is the ID of the port most recently operated on by a screen or accept statement.

If there are no fields, `last_screen_field` returns the null field ID (-32768).

You can retrieve the IDs of all fields of a form (in reverse of display order) by invoking `last_screen_field`, followed by a series of invocations of `prev_screen_field`. See the example in the description of the `prev_screen_field` function later in this section.

```
next_changed_field (field_id [ ,form_id [ ,port_id ] ] )
```

This function returns the ID of the field following the specified field in the internal changed-field list.

The value of *field_id* must be the field ID of a changed field in the form; typically, this value is the result returned by either *first_changed_field* or a previous invocation of *next_changed_field*. The value of *form_id*, if given, must be a valid form ID as returned by the *perform screen initialization* statement. If *form_id* is omitted or is less than 0, the ID of the most recently referenced form is used. The value of *port_id*, if given, must be the integer ID of the port on which the form is to be displayed. If *port_id* is omitted or is less than 0, the default is the ID of the port most recently operated on by a *screen* or *accept* statement.

The internal changed-field list contains all fields that were changed in the most recent display of the form. The order of the list is undefined.

If the *field_id* value you give to *next_changed_field* is the last changed field in the list, the function returns the null field ID (-32768).

You can retrieve the IDs of all changed fields in a form by invoking *first_changed_field*, followed by a series of invocations of *next_screen_field*. The following example illustrates this usage.

```
%replace NULL_FIELD_ID by -32768

01 field_id      comp-4.
   .
   .
   .

   move !first_changed_field () to field_id.

   perform process-field until field_id equal NULL_FIELD_ID.
   .
   .
   .
process-field.
   .
   .      /* Process this field. */
   .

   move !next_changed_field (field_id) to field_id.
```

`next_screen_displaytype (display_type_id [,port_id])`

This function returns the next highest display-type ID from the list of currently used global display types.

The value of *display_type_id* must be the ID of a global display type; typically, this value is the result returned by either `first_screen_displaytype` or a previous invocation of `next_screen_displaytype`. The value of *port_id*, if given, must be the integer ID of the port on which the form is to be displayed. If *port_id* is omitted or is less than 0, the default is the ID of the port most recently operated on by a screen or accept statement.

The list of global display types is sorted numerically.

If the *display_type_id* you give to the `next_screen_displaytype` function is the last display type in the list, the function returns the null display-type ID (-32768).

You can retrieve all global display-type IDs currently in use (in numerical order) by invoking `first_screen_displaytype`, followed by a series of invocations of `next_screen_displaytype`. The following example illustrates this usage.

```
%replace NULL_DISPLAY_TYPE_ID by -32768

01 display_type_id comp-4.
   .
   .
   .

move !first_screen_displaytype ( ) to display_type_id.

perform process-display-type
until display_type_id equal NULL_DISPLAY_TYPE_ID.
   .
   .
   .

process-display-type.
   .
   .    /* Process this display type. */
   .

move !next_screen_displaytype (display_type_id)
to display_type_id.
```

```
next_screen_field (field_id [ ,form_id [ ,port_id ] ] )
```

This function returns the field ID that follows the specified field ID in the display list.

The value of *field_id* must be the field ID of a field in the display list. The value of *form_id*, if given, must be a valid form ID as returned by the perform screen initialization statement. If *form_id* is omitted or is less than 0, the ID of the most recently referenced form is used. The value of *port_id*, if given, must be the integer ID of the port on which the form is to be displayed. If *port_id* is omitted or is less than 0, the default is the ID of the port most recently operated on by a screen or accept statement.

If the *field_id* value you give to *next_screen_field* is the last field in the list, the function returns the null field ID (-32768).

The list of fields includes all fields the program can manipulate, plus all fields created by the Forms Processor for background text.

You can retrieve the IDs of all fields of a form (in display order) by invoking *first_screen_field*, followed by a series of invocations of *next_screen_field*. The following example illustrates this usage.

```
%replace NULL_FIELD_ID by -32768

01 field_id      comp-4.
    .
    .
    .

    move !first_screen_field () to field_id.

    perform process-field until field_id equal NULL_FIELD_ID.
    .
    .
    .
process-field.
    .
    .      /* Process this field. */
    .

    move !next_screen_field (field_id) to field_id.
```

```
prev_screen_displaytype (display_type_id [ ,port_id ] )
```

This function returns the display-type ID that precedes the specified display-type ID on the list of currently allocated global display types.

The value of *display_type_id* must be the ID of a global display type; typically, this value is the result returned by either *last_screen_displaytype* or a previous invocation of *prev_screen_displaytype*. The value of *port_id*, if given, must be the integer ID of the port on which the form is to be displayed. If *port_id* is omitted or is less than 0, the default is the ID of the port most recently operated on by a screen or accept statement.

The list of global display types is sorted numerically.

If the *display_type_id* you give to the *next_screen_displaytype* function is the last display type in the list, the function returns the null display type ID (-32768).

You can retrieve all global display-type IDs currently in use (in reverse numerical order) by invoking *last_screen_displaytype*, followed by a series of invocations of *prev_screen_displaytype*. The following example illustrates this usage.

```
%replace NULL_DISPLAY_TYPE_ID by -32768

01 display_type_id comp-4.
   .
   .
   .

   move !last_screen_displaytype () to display_type_id.

   perform process-display-type
      until display_type_id equal NULL_DISPLAY_TYPE_ID.
      .
      .
      .

process-display-type.
   .
   .   /* Process this display type. */
   .

   move !prev_screen_displaytype (display_type_id)
      to display_type_id.
```

```
prev_screen_field (field_id [ ,form_id [ ,port_id ] ] )
```

This function returns the field ID that precedes the specified field ID in the display list.

The value of *field_id* must be the valid field ID of a field in the display list; typically, this value is the result returned by either *last_screen_field* or a previous invocation of *prev_screen_field*. The value of *form_id*, if given, must be a valid form ID as returned by the perform screen initialization statement. If *form_id* is omitted or is less than 0, the ID of the most recently referenced form is used. The value of *port_id*, if given, must be the integer ID of the port on which the form is to be displayed. If *port_id* is omitted or is less than 0, the default is the ID of the port most recently operated on by a screen or accept statement.

If the *field_id* value you give to *prev_screen_field* is the first field in the list, the function returns the null field ID (-32768).

You can retrieve the IDs of all fields of a form (in reverse of display order) by invoking *last_screen_field*, followed by a series of invocations of *prev_screen_field*. The following example illustrates this usage.

```
%replace NULL_FIELD_ID by -32768

01 field_id      comp-4.
   .
   .
   .

   move !last_screen_field () to field_id.

   perform process-field until field_id equal NULL_FIELD_ID.
   .
   .
   .
process-field.
   .
   .    /* Process this field. */
   .

   move !prev_screen_field (field_id) to field_id.
```

`screen_field_position (row, column [,form_id [,port_id]]`

This function returns the field ID of the nearest input field that is located at or before the position specified by *row* and *column*.

The value of *row* must be an integer indicating a line of the screen. The value of *column* must be an integer indicating a column of the screen. The value of *form_id*, if given, must be a valid form ID as returned by the perform screen initialization statement. If *form_id* is omitted or is less than 0, the ID of the most recently referenced form is used. The value of *port_id*, if given, must be the integer ID of the port on which the form is to be displayed. If *port_id* is omitted or is less than 0, the default is the ID of the port most recently operated on by a screen or accept statement.

If the position specified by *row* and *column* is the start of an input field, the function returns the field ID for that field. If *row* and *column* do not specify the start of an input field, but an input field is located on that row to the left of *column*, the function returns the field ID of that field. If no field meets that criteria, then the function returns the field ID of the nearest input field on a previous row.

If no input field appears at or after the specified position, the function returns the null field ID (-32768).

Chapter 16:

Statements

This chapter describes the statements used to manipulate screen forms. Table 16-1 lists these statements.

Table 16-1. The FMS Statements

Statement	Purpose
accept	Obsolete.
perform screen delete	Deletes a field or display type from the display list.
perform screen discard	Frees the storage space used by a form.
perform screen initialization	Sets up the display list for a form.
perform screen input	Writes a form to the terminal and accepts input from the user.
perform screen inquire	Obtains information about the display list.
perform screen output	Writes a form to the terminal; does not accept input from the user.
perform screen save	Stores a form to be retrieved later.
perform screen update	Alters the display list; does not change the user's screen.

The accept Statement

Purpose

The accept statement is obsolete. It displays or redisplay a screen form and accepts input from the user.

Syntax

```
accept [ 'form_name' [ into (field_values)
                    [ update (field_values) ] ] ] [ with form_option... ]
      [ using field_description... ] [ giving dt_description... ].
```

Operands

► *form_option*

The following form options are valid in the accept statement:

beep	into	portid
clear	keyused	putcursor
datastates	maskkeys	redisplay
displaytypes	message	status
form	modes	timeout
formid	nextcursor	update
functionkey	options	
getcursor	origin	

The form options are described in Chapter 5, “Form Options.”

► *field_description*

The description of a field in the form. The following field options are valid in the accept statement:

array	initial	redisplayfield
center	left	required
cycle	length	right
datastate	mode	update
displaytype	picture	validate
given	position	window
help	range	

For information on field descriptions, see Chapter 6, “Field Descriptions.”

Explanation

When an accept statement is executed, the Forms Processor does the following:

- For the initial display of a form, the Forms Processor reads the form definition as given by the predefined form (if any) and the options of the accept statement itself. From this definition, it initializes the display list. The Forms Processor then displays the form on the user's terminal display.

For the redisplay of a form, the Forms Processor updates the existing display list with information given in the accept statement. The Forms Processor then updates the existing form display on the user's terminal.

- The Forms Processor accepts forms input from the user.
- The Forms Processor handles user forms edit requests.
- The Forms Processor accepts and validates the data entered by the user.
- The Forms Processor converts each field input value to the data type of the associated field input variable and assigns that value to the variable in the program.
- The Forms Processor displays help messages, if requested by the user.

See Appendix A, "The accept Statement," for more information on the accept statement.

Example

The following is an example of an accept statement that references a predefined form.

```

data division.
working-storage section.

01 fields.
   copy 'name_address.incl.cobol'.

01 error_code           comp-4.
01 key_code             comp-4.
01 redisplay_switch     comp-4.

procedure division.

   accept 'name_address' into (fields)
      with keyused (key_code) status (error_code)
      redisplay (redisplay_switch).
```

In the following example, field descriptions are used to modify the predefined form. The form contains five predefined fields: name, street, city, state, and phone.

```
data division.
working-storage section.

    copy 'name_address_ids.incl.cobol'.

01 fields.
    copy 'name_address.incl.cobol'.

01 error_code          comp-4.
01 key_code            comp-4.
01 redisplay_switch    comp-4.

procedure division.

    accept 'name_address'
        with keyused (key_code) status (error_code)
        redisplay (redisplay_switch)
        using
            field (name_id) update (name of fields)
            field (street_id) update (street of fields)
            field (city_id) update (city of fields)
            field (state_id) update (state of fields)
            field (phone_id) update (phone of fields).
```

The following is an example of an accept statement with no predefined form.

```
%replace name_title_id          by 1
%replace name_id                 by 2
%replace street_title_id        by 3
%replace street_id              by 4
%replace city_title_id          by 5
%replace city_id                by 6
%replace state_title_id         by 7
%replace state_id               by 8
%replace phone_title_id         by 9
%replace phone_id               by 10
%replace name_address_max_ids   by 10

%replace UNDERLINED             by 8
%replace LOW_INTENSITY          by 16
%replace HIGH_INTENSITY         by 32
```

(Continued on next page)

(Continued)

01 fields.

```

    20 name      pic x(32).
    20 street    pic x(32).
    20 city      pic x(32).
    20 state     pic x(32).
    20 phone     pic x(12).

```

```

01 error_code    comp-4.
01 key_code      comp-4.
01 redisplay_switch comp-4.

```

procedure division.

accept

```

    with keyed (key_code) status (error_code)
    redisplay (redisplay_switch)

```

using

```

    field (name_title_id) position (1,2) length (5)
    mode (LOW_INTENSITY) 'NAME:',
    field (name_id) position (1,13) length (32)
    mode (HIGH_INTENSITY + UNDERLINED) update (name of fields),
    field (street_title_id) position (2,2) length (7)
    mode (LOW_INTENSITY) 'STREET:',
    field (street_id) position (2,13) length (32)
    mode (HIGH_INTENSITY + UNDERLINED) update (street of fields),
    field (city_title_id) position (3,2) length (5)
    mode (LOW_INTENSITY) 'CITY:',
    field (city_id) position (3,13) length (32)
    mode (HIGH_INTENSITY + UNDERLINED) update (city of fields),
    field (state_title_id) position (4,2) length (6)
    mode (LOW_INTENSITY) 'STATE:',
    field (state_id) position (4,13) length (32)
    mode (HIGH_INTENSITY + UNDERLINED) update (state of fields),
    field (phone_title_id) position (5,2) length (6)
    mode (LOW_INTENSITY) 'PHONE:',
    field (phone_id) position (5,13) length (12)
    mode (HIGH_INTENSITY + UNDERLINED)
    picture ('999-999-9999') update (phone of fields).

```

The perform screen delete **Statement**

Purpose

The perform screen delete statement deletes fields and display types from the display list.

Syntax

```
perform screen delete
{
  [ portid (port_id) ] [ formid (form_id) ] [ status (status_code) ]
  {
    field (field_id)
    displaytype (display_type_id)
  }
  [ ,field (field_id)
    ,displaytype (display_type_id) ] ...
```

You can give the portid, formid, and status options in any order.

Operands

► *port_id*

The ID of the port associated with the form display list from which fields and display types are to be deleted. For more information, see the description of the portid form option in Chapter 5, “Form Options.”

► *form_id*

The form ID of the form from which fields are to be deleted. A form ID is obtained from the perform screen initialization statement. For more information, see the description of the formid form option in Chapter 5, “Form Options.”

► *status_code*

A returned status code. For more information, see the description of the status form option in Chapter 5, “Form Options.”

► *field_id*

The integer field ID of a field to be removed from the display list.

► *display_type_id*

The integer display-type ID of a display type to be removed from the display list.

Explanation

The perform screen delete statement removes the specified fields and display types from the display list. The statement does not perform any I/O to the user's terminal. Therefore, field deletions are not reflected on the screen until the next perform screen output or perform screen input statement is executed.

If you are deleting fields, the *formid* option specifies the form from which the fields are to be deleted. If you omit the *formid* option, fields are deleted from the most recently referenced form.

You can delete a display type only if it is not referenced by any existing field in any active form. You can delete predefined, temporary, and programmer-defined global display types. You cannot delete reserved global display types (those with display-type IDs in the range 0 to 10, inclusive).

The perform screen discard Statement

Purpose

The perform screen discard statement discards a form that has previously been saved with the perform screen save statement.

Syntax

```
perform screen discard [ with [ portid (port_id) ] ]  
                        [ formid (form_id) ]  
                        [ status (status_code) ] .
```

You can give the portid, formid, and status options in any order.

Operands

► *port_id*

The ID of the port associated with the form to be discarded. For more information, see the description of the portid form option in Chapter 5, “Form Options.”

► *form_id*

The form ID of the form to be discarded. A form ID is obtained from the perform screen initialization statement. For more information, see the description of the formid form option in Chapter 5, “Form Options.”

► *status_code*

A returned status code. For more information, see the description of the status form option in Chapter 5, “Form Options.”

Explanation

The perform screen discard statement decrements the reference count associated with an active form. When the reference count reaches zero, the form is discarded. This allows user heap space to be reused.

You can explicitly increment the reference count for a form by executing the perform screen save statement.

For information on how reference counts are handled, see Chapter 12, “Form Caching.”

The perform screen initialization Statement

Purpose

The perform screen initialization statement initializes a form display list.

Syntax

```
perform screen initialization {
    form_option...
    field_description
    display_type_description
}
[ ,field_description
,display_type_description ] ....
```

Operands

► *form_option*

The following form options are valid in the perform screen initialization statement:

datastates	maskkeys	portid
displaytypes	message	putcursor
form	nextcursor	status
formid	options	timeout
into	origin	update

The form options are described in Chapter 5, “Form Options.”

► *field_description*

The description of a field in the form. In the perform screen initialization statement, field descriptions can either define or modify fields. The following field options are valid in the perform screen initialization statement:

array	initial	unshift
datastate	length	update
displaytype	position	window
help	shift	

For information on field descriptions, see Chapter 6, “Field Descriptions.”

► *display_type_description*

The description of a display type. In the perform screen initialization statement, display-type descriptions can define or modify display types. The following display-type options are valid in the perform screen initialization statement:

action	range
charset	validate
cycle	visual
picture	

For information on display-type descriptions, see Chapter 7, “Display Types.”

Explanation

The perform screen initialization statement sets up the internal display list for a form and is always the first FMS statement to operate on a form. It does not perform any I/O to the user's terminal. You can subsequently use the perform screen input or perform screen output statement to display the form.

The following form options provide input to the perform screen initialization statement and affect the displayed form:

form	
maskkeys	(if the form is alterable by accept)
message	(if the form is alterable by accept)
options	
origin	
portid	
putcursor	
timeout	
update	

The perform screen initialization statement returns information in the following form options:

datastates
displaytypes
formid
into
nextcursor

You can use the returned values in subsequent perform screen output and perform screen input statements.

To reference a predefined form, you must include the form option in the perform screen initialization statement. If the application references more than one form, include the formid option to obtain a unique identifier for each form.

The `into` option allows you to obtain initial values for the variables that you will use in subsequent `perform screen output` and `perform screen input` statements.

If you use the `nextcursor` form option, the value returned is the ID of the first field in the form (that is, the default value for the `putcursor` form option).

The default value for the `portid` option is the user's terminal port. If you omit the `origin` option, the screen is cleared before initial display of the form. If you reference a predefined form, the default values for the `maskkeys`, `message`, `putcursor`, and `timeout` options are taken from the form definition.

The perform screen input **Statement**

Purpose

The perform screen input statement writes a form to the screen and waits for input from the user.

Syntax

$$\text{perform screen input } \left\{ \begin{array}{l} \text{form_option...} \\ \text{field_description} \\ \text{display_type_description} \end{array} \right\} \left[\begin{array}{l} \text{,field_description} \\ \text{,display_type_description} \end{array} \right] \dots$$

Operands

► *form_option*

The following form options are valid in the perform screen input statement:

beep	keyused	portid
datastates	maskkeys	putcursor
displaytypes	message	status
formid	nextcursor	timeout
functionkey	options	update
getcursor		

The form options are described in Chapter 5, “Form Options.”

► *field_description*

The description of a field in the form. In the perform screen input statement, field descriptions can either define or modify fields. The following field options are valid in the perform screen input statement:

array	initial	unshift
datastate	length	update
displaytype	position	window
help	shift	

For information on field descriptions, see Chapter 6, “Field Descriptions.”

► *display_type_description*

The description of a display type. In the perform screen input statement, display-type descriptions can define or modify display types. The following display-type options are valid in the perform screen input statement:

action	range
charset	validate
cycle	visual
picture	

For information on display-type descriptions, see Chapter 7, “Display Types.”

Explanation

The perform screen input statement writes the form to the user's terminal and waits for input from the user. The Forms Processor allows the user to move the cursor among the input fields of the form and to change the input values.

The following form options are read by the Forms Processor and affect the display of the form:

beep	
datastates	(if the COPY_DATASTATE options switch is true)
displaytypes	
formid	
maskkeys	
message	
options	
portid	
putcursor	
timeout	
update	(unless the NO_COPY_UPDATE options switch is true)

Information is returned in the following form options:

datastates	getcursor	status
displaytypes	keyused	update
functionkey	nextcursor	

Control transfers to the statement following the perform screen input statement when the user either submits or cancels the form, when the timeout period for the form is exceeded, or when a trap occurs. You can determine what action caused control transfer by examining the output of the status form option and either the keyused or functionkey form option. The getcursor form option returns the location of the cursor when the form was submitted or canceled. If a trap occurs, the nextcursor option returns the location where the cursor would be if the trap had not occurred.

The values of the input fields are returned in the update option, unless the user cancels the form.

The perform screen inquire Statement

Purpose

The perform screen inquire statement returns information about the current display list.

Syntax

```
perform screen inquire {
    form_option...
    field_description
    display_type_description
}
[ ,field_description
,display_type_description ] ...
```

Operands

► *form_option*

The following form options are valid in the perform screen inquire statement:

datastates	max_displaytype_id	origin
displaytypes	max_field_id	portid
form	name	status
formid	nextcursor	timeout
maskkeys	options	

The form options are described in Chapter 5, “Form Options.”

► *field_description*

The description of a field in the display list. Field descriptions in an perform screen inquire statement return information about the field. The following field options are valid in the perform screen inquire statement:

array	initial	shift
datastate	length	unshift
displaytype	name	update
help	position	window

For information on field descriptions, see Chapter 6, “Field Descriptions.”

► *display_type_description*

The description of a display type used in the display list. Display-type descriptions in the perform screen inquire statement return information about the display type. The following display-type options are valid in the perform screen inquire statement:

action	name	validate
charset	picture	visual
cycle_array	range	

For information on display-type descriptions, see Chapter 7, “Display Types.”

Explanation

The perform screen inquire statement returns information about the display list.

The portid and formid options specify for which form information is to be returned. The status option returns a status code for the statement. All other form options return information about the display list.

With the exception of the shift and unshift options, all operands within field options return information about the field in a perform screen inquire statement. The shift and unshift options are input-only in all contexts.

In a perform screen inquire statement, all operands within display-type options receive information about the display type.

The perform screen output **Statement**

Purpose

The perform screen output statement writes the form to the user's terminal, but does **not** wait for input from the user.

Syntax

```
perform screen output {
    form_option...
    field_description
    display_type_description
}
[ ,field_description
  ,display_type_description ] ...
```

Operands

► *form_option*

The following form options are valid in the perform screen output statement:

beep	maskkeys	putcursor
datastates	message	status
displaytypes	options	timeout
formid	portid	update

The form options are described in Chapter 5, "Form Options."

► *field_description*

The description of a field in the form. In the perform screen output statement, field descriptions can either define or modify fields. The following field options are valid in the perform screen output statement:

array	initial	unshift
datastate	length	update
displaytype	position	window
help	shift	

For information on field descriptions, see Chapter 6, "Field Descriptions."

► *display_type_description*

The description of a display type. In the perform screen output statement, display-type descriptions can define or modify display types. The following display-type options are valid in the perform screen output statement:

action	range
charset	validate
cycle	visual
picture	

For information on display-type descriptions, see Chapter 7, “Display Types.”

Explanation

The perform screen output statement writes the form to the user’s terminal. It does **not** wait for input from the user.

The following form options are read by the Forms Processor and affect the display of the form:

beep	
datastates	(if the COPY_DATASTATE options switch is true)
displaytypes	
formid	
maskkeys	
message	
options	
portid	
putcursor	
timeout	
update	(unless the NO_COPY_UPDATE options switch is true)

Information is returned in the following form options:

datastates
displaytypes
status
update

The maskkeys, putcursor, and timeout options take effect on the next perform screen input statement (unless they are overridden by options in that statement or an intervening statement).

The perform screen save **Statement**

Purpose

The perform screen save statement saves the display list of a form for later retrieval.

Syntax

```
perform screen save [ portid (port_id)
                    [ formid (form_id)
                    [ status (status_code) ] .
```

You can give the portid, formid, and status options in any order.

Operands

► *port_id*

The ID of the port associated with the form. For more information, see the description of the portid form option in Chapter 5, "Form Options."

► *form_id*

The ID of the form to be saved. A form ID is obtained from the perform screen initialization statement. For more information, see the description of the formid form option in Chapter 5, "Form Options."

► *status_code*

A returned status code. For more information, see the description of the status form option in Chapter 5, "Form Options."

Explanation

The perform screen save statement increments the internal reference count for a form. As long as this reference count is greater than 0, the form's display list is saved in the user heap.

You should save a form if you might be using the form again in the application but currently want to use the screen for other forms. You can use the saved form later without executing another perform screen initialization statement. This provides for better performance because initializing a form is relatively inefficient.

You can explicitly decrement the internal reference count for a form by executing the perform screen discard statement.

For information on how reference counts are handled, see Chapter 12, “Form Caching.”

The perform screen update **Statement**

Purpose

The perform screen update statement changes one or more values or attributes in a form's display list. The changes appear on the screen when a subsequent perform screen output or perform screen input statement is executed.

Syntax

```
perform screen update {
    form_option...
    field_description
    display_type_description
}
[ ,field_description
,display_type_description ] ...
```

Operands

► *form_option*

The following form options are valid in the perform screen update statement:

beep	maskkeys	putcursor
datastates	message	status
displaytypes	options	timeout
formid	portid	update

The form options are described in Chapter 5, "Form Options."

► *field_description*

The description of a field in the form. A field description in the perform screen update statement can define or modify a field. The following field options are valid in the perform screen update statement:

array	initial	unshift
datastate	length	update
displaytype	position	window
help	shift	

For information on field descriptions, see Chapter 6, "Field Descriptions."

► *display_type_description*

The description of a display type used in the form. A display-type option in the perform screen update statement creates or modifies a display type. The following display-type options are valid in the perform screen update statement:

action	range
charset	validate
cycle	visual
picture	

For information on display-type descriptions, see Chapter 7, “Display Types.”

Explanation

The perform screen update statement modifies the display list, but does not change the screen appearance.

Use the perform screen update statement to make changes to the display list that do not need to be immediately flushed to the terminal screen. For example, you can use the perform screen update statement to add fields and display types to the display list, or to modify existing fields and display types. If you want changes to be reflected immediately, use the perform screen output or perform screen input statement instead.

You **cannot** use the perform screen update statement to delete a field from the display list. Use the perform screen delete statement for that purpose.

If you specify the displaytypes form option, the perform screen update statement reads the table of display types and applies the values to the form’s display list.

Appendix A:

The accept Statement

The accept statement is an obsolete FMS statement that is supported for existing applications. New applications should use the screen statements.

The syntax of the accept statement is similar to the syntax of the screen statements. The accept statement can include the form options, field descriptions, and display-type descriptions. For information on the syntax of the accept statement, see Chapter 16, "Statements." For information about each option allowed on the statement, see Chapter 5, "Form Options," Chapter 6, "Field Descriptions," and Chapter 7, "Display Types."

The old-style accept statement can include only form options and field descriptions. In the old-style statement, those attributes included in display types and data states are handled by field modes and several obsolete field options.

Several features are unique to the accept statement or are handled differently for the accept statement than for screen statements. To program with the accept statement, you must be familiar with the way the accept statement handles the following:

- display lists
- initial display versus redisplay
- the update form option versus the into form option
- field modes
- field-value justification
- obsolete field options
- field output values.

These issues are discussed in the following sections.

The Display List

In old-style applications, multiple accept statements cannot reference a single display list. The Forms Processor assumes that each accept statement in a program refers to a unique form.

If an old-style program contains two or more accept statements, the Forms Processor creates a display list for each of them, even if they reference the same predefined form. However, if the same accept statement is executed twice, it can use the same display list each time by using the `redisplay` form option.

Because of this behavior, if an old-style program displays the same form twice without intervening screen I/O, it is most efficient to re-execute the same accept statement for the second display. This allows the Forms Processor to use the same display list. Old-style programs are often designed to loop back to the accept statement for the redisplay. Other programs put the accept statement in a separate procedure, and invoke that procedure whenever the form must be displayed.

Note: When an old-style form is canceled by the user, the display list is discarded.

The following section explains how to handle initial displays and redisplays.

Initial Display and Redisplay

The accept statement can perform two types of form displays: initial displays and redisplays.

For an initial display, the Forms Processor does the following:

- discards an existing display list, if any
- creates and initializes the display list
- writes the entire form to the screen
- accepts input from the user and returns control to the program.

For a redisplay, the Forms Processor does the following:

- updates the existing display list
- writes any changes to the screen
- accepts input from the user and returns control to the program.

Note that a redisplay is more efficient than a second initial display in two ways. For a redisplay, the display list does not have to be re-created, and the entire form does not need to be rewritten to the screen.

You can perform a redisplay only if the following conditions are true.

- The accept statement has already been executed at least once for initial display.
- The form still appears on the user's screen. That is, no other form has been displayed in the same window, and no other input or output that would disrupt the form has been performed.
- The form has not been canceled in its most recent display.

If you wish to display a form again after another form has been displayed in the window, or after other I/O has interfered with the form display, you must perform an initial display of the form.

The redisplay form option of the accept statement indicates whether the Forms Processor is to perform an initial display of the form or a redisplay. If the value of the operand of the redisplay option is 0, or if the option is omitted, an initial display of the form is performed. If the operand value is non-zero, a redisplay is performed.

The first time an accept statement is executed, an initial display of the form must be performed, not a redisplay. If the statement includes the redisplay option, the value of its operand must be 0.

An accept statement that performs an initial display is analogous (but not exactly equivalent) to the combination of a perform screen initialization statement followed by a perform screen input statement. An accept statement that performs a redisplay is analogous to a subsequent perform screen input statement.

The following fragment illustrates one method for constructing a loop that displays and redispays a form.

```

move 0 to redisplay_switch.

perform loop until code not equal 0.

loop.
    accept form (employee_info) into (employee_fields)
        redisplay (redisplay_switch) status (code).
        .
        .
        .
    move 1 to redisplay_switch.
```

You can also put the accept statement into a procedure that is called from various points in the program. The following fragment is an example of such a procedure.

```
identification division.  
program-id. display_form.  
  
data division.  
linkage section.  
  
    01 p_redisplay_switch    comp-4.  
    01 p_code                comp-4.  
  
procedure division using p_redisplay_switch, p_code.  
  
    accept form (employee_info) into (employee_fields)  
        redisplay (p_redisplay_switch) status (p_code).  
  
exit program.
```

Note that you can pass a non-zero value to `p_redisplay_switch` only if the conditions necessary for redisplay are met.

The field modes and the `into` and `update` form options are also treated differently on redisplay than on initial display. These differences are discussed in the following two sections.

The `into` and `update` Form Options

An `accept` statement can include either the `into` form option or the `update` form option to return field values. The distinction between the options is the manner in which field output values are determined on initial display.

Generally, if the `accept` statement includes the `into` form option, any initial field values specified in the Forms Editor are used as field output values on initial display. If the `accept` statement includes the `update` option, initial values specified in the Forms Editor are ignored.

On redisplay, the current values of the field-value variables specified in the `into` or `update` option are displayed in the fields, unless you use the `redisplayfield` field option. For information on the `redisplayfield` option, see Chapter 6, “Field Descriptions.”

For the complete sets of rules that determine field output values, see the section “Field Output Values” later in this appendix.

If you use either the `into` or `update` form option in an `accept` statement, most field descriptions are invalid in that statement. The only field descriptions that are allowed are those of the following format:

```
field (field_id) mode (mode_expression)
```

The mode field option is discussed in the following section of this appendix and in Chapter 6, "Field Descriptions."

For further information on the into and update form options, see Chapter 5, "Form Options."

Field Modes

The old-style accept statement does not use display types or data states. Instead, most of the attributes now specified by display-type visual and action switches and some of the attributes now specified by data-state switches are specified by a set of switches called the *field modes*. (For a complete mapping of old features to new features, see Appendix C, "Converting Old-Style Applications.")

A field's modes are stored as a two-byte integer. A series of switches are encoded within this integer as shown in Table A-1.

Table A-1. The Mode Switches

Bit	Mode Switch
1	BLANKED
2	BLINKING
4	INVERSE
8	UNDERLINED
16	LOW_INTENSITY
32	HIGH_INTENSITY
256	INPUT_DISABLED
512	NO_OVERLAY
1024	AUTO_TAB (or AUTO_TAB_TO_NEXT_FIELD)
2048	IMMEDIATE_RETURN (or TRAP_ON_FIELD_EXIT)
4096	NOT_EDITABLE (or DISAPPEARING_DEFAULT)
8192	TRAP (or TRAP_ON_FIELD_ENTRY)

All unused bits are reserved and must be set to 0.

The six low-order bits of the mode value have the same meanings as the six low-order bits of the display-type visual switches. The AUTO_TAB, IMMEDIATE_RETURN, and TRAP mode switches correspond to the AUTO_TAB_TO_NEXT_FIELD, TRAP_ON_FIELD_EXIT, and TRAP_ON_FIELD_ENTRY display-type action switches, respectively. The display-type visual and action switches are described in Chapter 7, "Display Types." The INPUT_DISABLED mode switch is the inverse of the INPUT_FIELD data-state switch. The NOT_EDITABLE

mode switch has the same meaning as the `DISAPPEARING_DEFAULT` data-state switch. The data-state switches are described in Chapter 8, “Data States.”

The meaning of the `NO_OVERLAY` mode switch is similar to the meaning of the `FORCE_INSERT_MODE` display-type action switch. With the old-style accept statement, the field justification determines whether a field is initially in insert mode or overlay mode. If the field is left-justified, the initial editing mode is insert. If the field is right-justified, the initial editing mode is overlay. The `NO_OVERLAY` mode switch is meaningful only for right-justified fields. It changes the initial editing mode to insert. The user can always change the initial editing mode of a field with the `(INSERT/OVERLAY)` key.

Initializing Field Modes

The `(MENU) F` form of the old-style Forms Editor includes options that specify the modes for each predefined field. For information on the old-style Forms Editor, see Appendix B, “The `edit_form` Command.”

If you create a field dynamically in an accept statement, you can specify the modes for that field in a mode field option within the field description. If you omit the mode field option from the field description, the default mode values are used: 0 for an input field, and 16 (low intensity) for an output field.

Note that if you want to dynamically create an input field with input initially disabled, you must explicitly specify the modes for the field with the `INPUT_DISABLED` switch set to true.

Modifying Field Modes

The mode field option specifies the modes for a specific field; the modes form option specifies the modes for each field in a form. The operand for the mode field option is a two-byte integer. The operand for the modes form option is a table of two-byte integers. This table is called the *modes table*.

On initial display, the modes form option is output-only; it returns the modes of each field in the form. On redisplay, the modes option is used for both input and output. This means that you can change the modes of a field on redisplay by altering the corresponding value in the modes table.

The mode field option is used only as input and only on initial display. This allows you to change the modes of a predefined field on initial display. The mode value you specify is applied to the initial display and is returned into the modes table given in the modes form option. On redisplay, the mode field option is ignored, and the field modes are read from the modes form option.

The mode field option is the only field option allowed in the accept statement when that statement includes either the update or into form option.

Modes Example

The following example illustrates the handling of field modes within a program.

```
identification division.
program-id. modes_values.
```

```
* The program displays a form called name_address. The form's
* fields are "name", "street", "city", "state", and "phone".
```

```
data division.
working-storage section.
```

```
%replace BLANKED          by 1
%replace BLINKING         by 2
%replace INVERSE          by 4
%replace UNDERLINED       by 8
%replace LOW_INTENSITY    by 16
%replace HIGH_INTENSITY   by 32
%replace INPUT_DISABLED   by 256
%replace NO_OVERLAY       by 512
%replace AUTO_TAB         by 1024
%replace IMMEDIATE_RETURN by 2048
%replace NOT_EDITABLE     by 4096
%replace TRAP             by 8192
%replace CANCEL           by -1
%replace FALSE            by 0
%replace TRUE             by 1
```

```
copy 'name_address_ids.incl.cobol'.
```

```
01 fields.
copy 'name_address.incl.cobol'.
```

```
01 redisplay_switch comp-4.
01 status_code      comp-4.
01 key_code         comp-4.
01 modes_table.
    02 mode_info     comp-4
        occurs NAME_ADDRESS_MAX_IDS times.
```

(Continued on next page)

(Continued)

procedure division.

* Prepare for initial form display.

move FALSE to redisplay_switch.

* Display and redisplay the form until user cancels.

perform display-form until ((key_code equal CANCEL)
or (status_code not equal 0)).

if status_code not equal 0 then
go to fatal-error.

.
.
.

display-form.

accept form (name_address) into (fields) keyused (key_code)
status (status_code) redisplay (redisplay_switch)
modes (modes_table),
field (PHONE_ID) mode (INPUT_DISABLED).

* Prepare for subsequent form displays. Subsequent
* displays are redispays with the phone field enabled.

move TRUE to redisplay_switch.
move HIGH_INTENSITY + UNDERLINED to mode_info(PHONE_ID).

fatal-error.

* Handle error.

.
.
.

exit program.

For further information on the modes form option, see Chapter 5, "Form Options."
For further information on the mode field option, see Chapter 6, "Field Descriptions."

Note: On initial display, all elements of an array field have the same mode value. You can specify this value in the Forms Editor or in a mode field option. However, the modes table contains a value for each element in the array field. You can change the mode value of a specific element of an array field on redisplay by changing the corresponding modes table value.

Field-Value Justification

In old-style accept statements, the left, right, and center field options indicate the justification of a field. Support of these options is device-dependent. Justification options that are not supported are ignored.

The left field option indicates that the field is to be left-justified. Specifying this option for a field is analogous to setting the `LEFT_JUSTIFY_FIELD_DATA` display-type visual switch to true for the field.

The right field option indicates that the field is to be right-justified. Specifying this option for a field is analogous to setting the `RIGHT_JUSTIFY_FIELD_DATA` display-type visual switch to true for the field.

The center field option is **not** always analogous to the `CENTER_FIELD_DATA` display-type visual switch. For fields defined as input or input (initially disabled), the center field option does **not** center output values in the field. Rather, it affects the way the Forms Processor treats leading spaces in field values. Normally, any leading space characters are trimmed from an output value before it is displayed in the field, and any leading space characters are trimmed from a field value before it is validated and returned to the program. If the center field option is specified for the field, leading spaces are not trimmed on output or input. The untrimmed value is left-justified in the field. This allows the program to center an output value in a field by adding spaces to the left of a value before displaying the form.

For output-only fields, the center field option is analogous to the `CENTER_FIELD_DATA` display-type visual switch.

Obsolete Field Options

In addition to the mode field option and the justification options discussed earlier in this appendix, the following field options are unique to the old-style accept statement.

- cycle (*value* [*,value*] ...) (See Table A-1)
- given (*values_count*)
- picture (*field_picture*)
- range (*low_bound*, *high_bound*)

- `redisplayfield` (*redisplay_field_switch*)
- `required`
- `validate` (*validation_entry*)

Of these, the `cycle`, `picture`, `range`, and `validate` options have been replaced by like-named display-type options. The `given` and `required` options have been replaced by the `FIELD_VALUE_GIVEN` and `REQUIRED_FIELD` data-state switches.

The `redisplayfield` option indicates what value to display in the field when the form is redisplayed. If the value of the operand of the `redisplayfield` option is false, the field's initial output value is displayed. If the operand of the `redisplayfield` option is true, or if the option is not given, the current value of the field-value variable is displayed in the field.

For more information on field options, see Chapter 6, "Field Descriptions."

Field Output Values

The `accept` statement has three general formats, depending on its use. The formats are as follows:

- An `accept` statement that references a predefined form and includes the `into` or `update` form option. Such an `accept` statement cannot include field descriptions except those of the following format:

`field` (*field_id*) `mode` (*mode_expression*)

- An `accept` statement that references a predefined form, but does not include the `into` or `update` form option. Such an `accept` statement must include a field description with the `update` field option for each input field.
- An `accept` statement that does not reference a predefined form. Such an `accept` statement must include a field description for each field and cannot include an `into` or `update` form option.

For examples of these formats, see the description of the `accept` statement in Chapter 16, "Statements."

The rules the Forms Processor uses to determine what value to display in each field differ for the three types of `accept` statement.

Note: In all cases, if the initial output value derived for a cycle field is not in the cycle list, then the first value in the cycle list is used instead, provided it is not the null field value.

Predefined Form with the into or update Form Option

This subsection describes the rules that determine the field output value when the accept statement references a predefined form and includes the into or update form option.

Initial Display. On the initial display of a form, the following rules apply if the accept statement includes the into form option.

1. If an initial value for the field is specified in the Forms Editor, then that initial value is displayed in the field.
2. If Rule 1 does not apply and the field is output-only and the form is alterable by accept, then the value of the field-value variable given in the into option is displayed in the field.
3. If Rules 1 and 2 do not apply, then the field's null value is displayed.

If the accept statement includes the update form option rather than the into form option, then the following rules apply.

1. If the field is output-only and an initial value for the field is specified in the Forms Editor, then that initial value is displayed in the field.
2. If Rule 1 does not apply and the field is output-only and the form is not alterable by accept, then the field's null value is displayed.
3. If Rules 1 and 2 do not apply, then the value of the field-value variable specified in the update option is displayed.

Redisplay. On the redisplay of a form, whether the into or update form option is used, then the value of the field-value variable specified in that option is displayed in the field.

Predefined Form without the into or update Form Option

This subsection describes the rules that determine the field output value when the accept statement references a predefined form but does not include the into or update form option.

Initial Display. On initial display of a form, the following rules determine the output value of a field if the form is alterable by accept.

1. If the field description includes the `initial field` option, then the value specified in that option is displayed in the field.
2. If Rule 1 does not apply and an initial value is specified in the Forms Editor, then that initial value is displayed in the form.
3. If Rules 1 and 2 do not apply, then the value of the field-value variable specified in the `update field` option is displayed in the field.

If the form is not alterable by accept, then the following rules apply.

1. If an initial value for the field is specified in the Forms Editor, then that initial value is displayed in the field.
2. If Rule 1 does not apply and the field is output-only, then the null field value is displayed.
3. If Rules 1 and 2 do not apply and the field description does not include the `initial field` option, then the value of the field-value variable specified in the `update field` option is displayed in the field.
4. If Rules 1 through 3 do not apply, then the value specified in the `initial field` option is displayed in the field.

Redisplay. On the redisplay of a form, the following rules determine the output value for each field.

1. If the field description includes the `update field` option, then the value of the field-value variable specified in that option is displayed in the field.
2. If Rule 1 does not apply, then the value specified in the `initial field` option is displayed in the field.

No Predefined Form

This subsection describes the rules that determine the field output value when the `accept` statement does not reference a predefined form.

Initial Display. On the initial display of a form, the following rules determine the field output value.

1. If the field description includes the initial field option, then the value specified in that option is displayed in the field.
2. If Rule 1 does not apply, then the value specified in the update field option is displayed in the field.

Redisplay. On the redisplay of a form, the following rules determine the field output value.

1. If the field description includes the update field option, then the value specified in that option is displayed in the field.
2. If Rule 1 does not apply, then the value specified in the initial field option is displayed in the field.

)

)

)

)

)

Appendix B:

The edit_form Command

This appendix describes the old-style Forms Editor and the command used to invoke it, `edit_form`. The old-style Forms Editor defines forms that are compatible with the old-style `accept` statement.

For information on updating old-style forms to be compatible with the new forms software, see Appendix C, “Converting Old-Style Applications.”

Within the old Forms Editor, most of the editing and menu requests are the same as for the new Forms Editor. However, there are some differences. The Insert literal request, `(MENU) L`, is not available in the old Forms Editor. The Add/modify field request, `(MENU) F`, and the Set/modify form options request, `(MENU) S`, are substantially different in the old Forms Editor. These requests are described later in this appendix.

The edit_form Command

Purpose

The `edit_form` command invokes the old-style Forms Editor.

CRT Form

edit_form

input_path:

form_path:

-into:

no

-prefix:

no

-library:

accept_field_definitions

-edit:

yes

-backup:

yes

-force_write:

no

-basic:

no

-cobol:

no

-fortran:

no

-pascal:

no

-pl1:

no

-c:

no

-produce_symtab:

yes

Lineal Form

```
edit_form input_path
    [ form_path ]
    [-into ]
    [-prefix ]
    [-library field_definitions_directory_name ]
    [-no_edit ]
    [-no_backup ]
    [-force_write ]
    [-basic ]
    [-cobol ]
    [-fortran ]
    [-pascal ]
    [-pl1 ]
    [-c ]
    [-no_produce_symtab ]
```

Arguments

► *input_path*

Required

The path name of an input form definition file. If the file name you specify does not have the suffix *.form*, the command adds that suffix. If the file does not exist, the Forms Editor behaves as if the file exists but is empty.

► *form_path*

An option specifying the file to which the edited form definition is to be written. If *form_path* does not have the suffix *.form*, the command adds that suffix. If you do not specify a value for *form_path*, it defaults to a file in the current directory with the same name as the file specified in the *input_path* option. If the specified file does not exist when you write out the form, the Forms Editor creates it.

The form being edited is given the simple name of the output form definition file without the suffix *.form*. A form name should not exceed **15** characters; otherwise, the names of some automatically generated include files might exceed 32 characters and be truncated.

Note: Do not give a form the same name as the program that displays it — both a form and its related program require uniquely named object modules.

► -into

(CYCLE)

An option to create a field-values file for each programming language specified by the language options. The Forms Editor names the field-values file (an include file) *form_name.incl.language* and puts the file in the current directory. You can override this argument with the `PRODUCE_INT0` option of the Forms Editor (MENU) S request.

► -prefix

(CYCLE)

An option to add a prefix to each field identifier name in any field-IDs file that the Forms Editor generates. The default prefix is the name of the form followed by an underline. You can override this argument with the `PREFIX` option of the Forms Editor (MENU) S request.

If you choose the `-prefix` option, the Forms Editor also adds the prefix to each variable name in any VOS BASIC or VOS FORTRAN field-values file that it generates. The field-values files for other languages are not affected by this argument.

► -library *field_definitions_directory_name*

An option to specify a directory for storing and retrieving field definition files. The Forms Editor searches the directory for field definition files when you use the (MENU) R request and writes field definition files to the directory when you use the (MENU) E request. If you do not specify this option, the default value is a subdirectory of your current directory named `accept_field_definitions`. If the directory you specify, either directly or by default, does not exist when you issue a (MENU) E request, the Forms Editor creates the directory.

► -no_edit

(CYCLE)

An option to create new language include files and a new object module from an existing form definition file without editing the form. If you specify the `-force_write` option with this option, the Forms Editor also writes a new form definition file. By choosing the `-no_edit` option, you can run the Forms Editor in either a batch process or a started process. If you do not use the `-no_edit` argument, the Forms Editor reads the form definition file, displays a representation of the form, and lets you edit it.

► -no_backup

(CYCLE)

An option to specify that no backup file is created for the *input_path* file. If you do not use the `-no_backup` option, and the *input_path* and *form_path* files are in the same directory, then the Forms Editor renames the old file and gives it the name of the *input_path* file (including its suffix `.form`), with the suffix `.backup` added. The backup file is created each time you write out the form with the (MENU) W request; it replaces a previous backup file of the same name, if one exists.

► -force_write

(CYCLE)

An option to write a new form definition file (*form_name.form*) when you invoke `edit_form` with the `-no_edit` option. If you do not use the `-force_write` option, `-no_edit` produces the object module and specified include files only. Use `-force_write` with `-no_edit` to generate a `.backup` form file or to rename your form without re-editing it.

Note: Do not use the command `rename` to rename a form; object and include files must be renamed, and prefixes in include files need to be reassigned.

► -basic

(CYCLE)

An option to create VOS BASIC versions of the field-IDs file and the field-values file. If you do not use the `-basic` option, the Forms Editor does not create VOS BASIC versions of the files. You can override this argument with the `BASIC` option of the Forms Editor (MENU) S request.

► -cobol

(CYCLE)

An option to create VOS COBOL versions of the field-IDs file and the field-values file. If you do not use the `-cobol` option, the Forms Editor does not create VOS COBOL versions of the files. You can override this argument with the `COBOL` option of the Forms Editor (MENU) S request.

► -fortran

(CYCLE)

An option to create VOS FORTRAN versions of the field-IDs file and the field-values file. If you do not use the `-fortran` option, the Forms Editor does not create VOS FORTRAN versions of the files. You can override this argument with the `FORTRAN` option of the Forms Editor (MENU) S request.

► -pascal

(CYCLE)

An option to create VOS Pascal versions of the field-IDs file and the field-values file. If you do not use the `-pascal` option, the Forms Editor does not create VOS Pascal versions of the files. You can override this argument with the `PASCAL` option of the Forms Editor (MENU) S request.

► -pl1

(CYCLE)

An option to create VOS PL/I versions of the field-IDs file and the field-values file. If you do not use the `-pl1` option, the Forms Editor does not create VOS PL/I versions of the files. You can override this argument with the `PL/I` option of the Forms Editor (MENU) S request.



(CYCLE)

An option to create VOS C versions of the field-IDs file and the field-values file. If you do not use the -c option, the Forms Editor does not create VOS C versions of the files. You can override this argument with the c option of the Forms Editor (MENU) s request.



(CYCLE)

An option to produce a form object module without a run-time symbol table. Because the forms run-time symbol table is small, use the -no_produce_symtab option only if there is a shortage of virtual memory.

Explanation

An edit_form command invokes the Forms Editor. After you issue the edit_form command, your process is at *edit request level*. At edit request level, you can enter text or you can make a number of edit requests. Most of these requests are the same as the requests for the new Forms Editor, which are described in Chapter 4, “The Forms Editor.” Those requests that are different are discussed in this appendix.

If you give the path name of an existing input form definition file when you issue the edit_form command, the Forms Editor reads the file and displays a representation of the defined form.

The Forms Editor trims trailing spaces from all the values you enter into the editor’s request forms and from all lines you enter into the form you are constructing. It also deletes all empty lines from the bottom of the form. When you write the form definition file and the other files described previously, the files reflect these deletions.

If you choose the -into option when you invoke the Forms Editor, but do not specify any of the languages at that time, you can specify one or more languages using the Forms Editor requests. If you do not specify any of the language options in the command line or in the Forms Editor, the -into option is ignored.

If you choose the -into option, the -prefix option, or any of the language options (-basic, -cobol, -fortran, -pascal, -pl1, or -c) for a particular form, these options are saved in the forms definition file, and you do not have to respecify them in the (MENU) F form or in future invocations of the Forms Editor on that form.

Access Requirements

You need read access to a form definition file to read it; you need write access to a form definition file, include file, or object module to write it.

The Add/modify field Request

In the **new** Forms Editor, the form for the Add/modify field (**MENU** F) request specifies initial values for field options, data-state switches, and display-type options, as well as the data types of field-value variables. In the **old** Forms Editor, the **MENU** F form specifies initial values for field options and mode switches, and the data types of field-value variables.

When you invoke the Add/modify field request in the old Forms Editor, the following form appears on the screen.

--Field Options--			
FIELD NAME	<div style="background-color: black; width: 150px; height: 1.2em;"></div>		
FIELD TYPE	input		
INITIAL	_____		
PICTURE	_____		
VALIDATE	_____		
REQUIRED	no	EDITABLE yes	TRAP no
VALUE RESTRICTION	none	BASIC	\$= _____
JUSTIFICATION		COBOL display	_____
AUTO TAB	no	FORTRAN character*	_____
IMMEDIATE RETURN	no	PASCAL char array	_____
FIELD EDIT MODE	overlay	PL/1 char / pic	_____
LENGTH	_____	C char []	_____
POSITION	_____ / _____	field-values sequence	_____
ARRAY LAYOUT	_____ / _____	ROW SPACING _____ / COLUMN SPACING	1
HELP	_____		
<div style="display: flex; justify-content: space-between; padding: 0 10px;"> INTENSITY: high underline not inverse </div> <div style="display: flex; justify-content: space-between; padding: 0 10px;"> non blinking not blanked </div>			

Unlike the **MENU** F form in the new Forms Editor, the **MENU** F form in the old Forms Editor is not divided into field options and display-type options. However, most of the fields in the form are the same for both Forms Editors, although the arrangement of fields is different.

The following are the only fields in the old (MENU) F form of the old Forms Editor that differ from fields in the form for the new Forms Editor.

- FIELD TYPE
- INITIAL
- EDITABLE
- TRAP
- JUSTIFICATION
- IMMEDIATE RETURN
- FIELD EDIT MODE

These fields are discussed in this section. For information on the other fields in the (MENU) F form, see Chapter 4, “The Forms Editor.”

► FIELD TYPE (CYCLE) input, input (initially disabled), output only

This field is the same as the FIELD TYPE field in the (MENU) F form of the new Forms Editor, except that the name of the second cycle value is input (initially disabled) rather than output. If you designate a field as input (initially disabled), the field is created as an output field, but you can dynamically change it to an input field by setting the INPUT_DISABLED mode switch to false.

For information on field modes, see Appendix A, “The accept Statement.”

► INITIAL

This field is the same as the INITIAL field in the (MENU) F form of the new Forms Editor, except that it cannot be used for output-only fields and behaves differently for cycle fields. If you specify a value for the INITIAL field before defining the cycle list for the field, the initial value becomes the first value in the cycle list. If you specify a value for the INITIAL field after defining the cycle list, then if the INITIAL value is in the cycle list, that value is moved to the beginning of the list. If you specify an INITIAL value that is not in the cycle list, that value is rejected.

► EDITABLE (CYCLE) yes, no

This field has been replaced by the DISAPPEARING field in the (MENU) F form of the new Forms Editor. If you set EDITABLE to no, the NOT_EDITABLE mode switch is initially true. This means that the output value disappears when the user types a character in the first position of the field. If you set EDITABLE to yes (the default), the NO_EDITABLE mode switch is initially false. This means that the output value does not disappear when the user types a character in the first position of the field.

This feature is ignored by some device drivers.

For information on field modes, see Appendix A, “The accept Statement.”

► TRAP (CYCLE) no, yes

This field has been replaced by the TRAP ON FIELD ENTRY field in the (MENU) F form of the new Forms Editor. If you set TRAP to yes for a field, the TRAP mode bit for the field is initially true. This means that control returns to the program as soon as the user positions the cursor to that field. If you set TRAP to no (the default), the TRAP mode bit for the field is initially false.

This feature is ignored by some device drivers.

For information on field modes, see Appendix A, "The accept Statement."

► JUSTIFICATION (CYCLE) null, left, right, center

This field appears the same as the JUSTIFICATION field in the new Forms Editor. However, in the old Forms Editor, this field is highly device dependent. This field might be supported only for alphanumeric fields with no field picture. Some device drivers always right justify numeric fields and left justify alphanumeric fields with field pictures.

See also the discussion of field justification in Appendix A, "The accept Statement." In particular, note that the meaning of center justification differs for old-style applications.

► IMMEDIATE RETURN (CYCLE) no, yes

This field has been replaced by the TRAP ON FIELD EXIT field in the (MENU) F form of the new Forms Editor. If you set IMMEDIATE RETURN to yes, then the IMMEDIATE_RETURN mode switch for the field is initially true. This means that control returns to the application when the user moves the cursor out of the field. If you set IMMEDIATE RETURN to no (the default), then the IMMEDIATE_RETURN mode switch for the field is initially false.

This feature is ignored by some device drivers.

For information on field modes, see Appendix A, "The accept Statement."

► FIELD EDIT MODE (CYCLE) overlay, insert

This field has been replaced by the FORCE INSERT MODE and FORCE OVERLAY MODE fields in the new Forms Editor. If you set FIELD EDIT MODE to insert, the NO_OVERLAY mode switch for the field is initially true. If you set FIELD EDIT MODE to overlay (the default), the NO_OVERLAY mode switch for the field is initially false.

This feature is ignored by some device drivers.

For information on field modes, see Appendix A, "The accept Statement."

The Set/modify form options Request

In the old Forms Editor, when you issue the Set/modify form options request, **(MENU)** s, the following form appears on the screen.

-- Form Options -- for form_name

MASKKEYS	no	CURRENCY SYMBOL	\$
INITIAL DISPLAY	clear	decimal is period	
ALTERABLE BY ACCEPT	yes	BANK TELLER DECIMAL	no
PRODUCE INTO	yes		
PREFIX			
MESSAGE			
BEEP	no		
BASIC	no		
COBOL	no		
FORTRAN	no		
PASCAL	no		
PL/1	no		
C	no		
BACKGROUND MODE			
INTENSITY:	low	no underline	not inverse
		non blinking	not blanked
REQ FIELD MODE TOGGLES			
INTENSITY TOGGLE:	same intensity	same underlining	toggle inverse
		same blinking	same blanking

The **(MENU)** s form in the new Forms Editor contains all the fields that are in the old version, plus some new ones. The only field in the old **(MENU)** s form that is not identical to a field in the new version is the BANK TELLER DECIMAL field.

In the new Forms Editor, you can choose whether to use the bank teller decimal action for each field; the BANK TELLER DECIMAL field in the **(MENU)** s form of the new Forms Editor indicates only the default choice. In the old Forms Editor, you must choose whether to use the bank teller decimal action for all fields with numeric pictures or for none; the BANK TELLER DECIMAL field of the **(MENU)** s form specifies the choice for all numeric fields. For information on the bank teller decimal action, see the description of the BANK_TELLER_DECIMAL display-type action switch in Chapter 7, "Display Types."

For information on other **(MENU)** s fields, see Chapter 4, "The Forms Editor."

The Insert Literal Request

The old Forms Editor does not support the Insert literal request, **(MENU)** L. You cannot insert characters from supplemental character sets into the background text of old-style forms, unless the terminal provides a means of entering them directly.

)

)

)

)

)

Appendix C:

Converting Old-Style Applications

This appendix describes how to update an old-style application to use the new FMS features. Old-style applications can still be run, but converting to the new features provides increased functionality and, in some cases, better performance.

Updating an application involves converting predefined forms with the new Forms Editor and updating the application program to use screen statements instead of the `accept` statement. After the program is compiled, the new application must be bound with the new FMS run-time routines.

Converting Predefined Forms

Forms produced by the old Forms Editor are compatible only with the old FMS run-time routines. Forms produced by the new Forms Editor are compatible only with the new FMS run-time routines.

To convert an old-style form to a new-style form, edit it with the new Forms Editor. The form definition file produced by the new Forms Editor is a new-style file. All old-style form options and field options are automatically converted to the corresponding new-style options. For information on these options, see Appendix B, "The `edit_form` Command."

You **cannot** edit a new-style form definition file with the old Forms Editor. However, you can use a single old-style form definition file to create both old-style and new-style form object modules. If you use the `-no_edit` option of the `icss_edit_form` command, the Forms Editor creates a new-style form object module but does not change the form definition file. Therefore, you can subsequently use the `edit_form` command to create an old-style form object module from the same form definition.

Updating the Application Program

In general, you can replace an accept statement in a program with the combination of an perform screen initialization statement and either an perform screen input or perform screen output statement. If the program executes the accept statement several times to redisplay the form, you can usually structure the program so that the perform screen initialization statement is executed only once and the perform screen input statement is executed repeatedly. For example, in the following fragment from an old-style program, the loop displays and redisplay a form.

```
move 0 to code.
move 0 to redisplay_switch.

perform old-loop until code not equal 0.

old-loop.
    accept form (employee_info) into (employee_fields)
        redisplay (redisplay_switch) status (code).
        .
        .
        .
    move 1 to redisplay_switch.
```

To update the program, you can rewrite this loop as follows:

```
perform screen initialization form (employee_info) formid (emp_info_id)
    into (employee_fields) status (error_code).

if error_code not equal 0 then
    go to fatal-error.

perform new-loop until error_code not equal 0.

new-loop.

    perform screen input formid (emp_info_id) update (employee_fields)
        status (error_code).
    .
    .
    .

fatal-error.
    .
    .
    .
```

The into and update Options

In the preceding example, note that the into option in the accept statement is replaced by an into option in the perform screen initialization statement (to obtain initial field values) and an update option in the perform screen input statement (to set and receive field values on each display). If the accept statement uses the update option instead, you should omit the into option in the perform screen initialization statement, but still use the update option in the perform screen input statement.

The form and formid Options

The form form option is used in an accept statement to reference a predefined form. To reference a predefined form with the screen statements, include the form form option in the perform screen initialization statement only.

If you also include a formid form option in the perform screen initialization statement, a unique integer ID for the form is returned in that option. The formid option is output-only in the perform screen initialization statement, but is input-only in all other screen statements. In subsequent screen statements, you can use the formid option with the value returned in the perform screen initialization statement to reference the same form.

For further information on the form and formid options, see Chapter 5, "Form Options."

Obsolete Options

Certain accept statement options have become obsolete with the addition of new options. Table C-1 lists the obsolete options and indicates which new option replaces each.

Table C-1. Replacements for Old Form and Field Options

Old Option	New Option
Form Options	
clear	†
modes	displaytypes and datastates form options
redisplay	†
Field Options	
center	CENTER_FIELD_DATA and NOTRIM_FIELD_DATA_SPACES display-type action switches
cycle	cycle display-type option
given	FIELD_VALUE_GIVEN data-state switch
left	LEFT_JUSTIFY_FIELD_DATA display-type action switch
mode	displaytype and datastate field options
picture	picture display-type option
range	range display-type option
redisplayfield	†
required	REQUIRED_FIELD data-state switch
right	RIGHT_JUSTIFY_FIELD_DATA display-type option
validate	validate display-type option

† The clear and redisplay form options, and the redisplayfield field option have no direct equivalent in the new Forms Editor.

For information on form options, see Chapter 5, “Form Options.” For information on field options, see Chapter 6, “Field Descriptions.” For information on display-type options, see Chapter 7, “Display Types.”

The obsolete options are still supported for the old-style accept statement, but if you are updating an application to use the screen statements, you should replace the old options with the new options.

The new FMS run-time software has no equivalent for the clear, redisplay and redisplayfield options. This does not mean that the functionality is not supported. A window is always cleared before a form is displayed in it, unless you specify in the Forms Editor or with the origin form option that the form is to scroll. With the new run-time software, the Forms Processor determines whether a particular form display is an initial display or a redisplay.

Note that the old field modes have been replaced by both display types and data states. Some of the old mode switches have been replaced by data-state switches, while others have been replaced by display-type action or visual switches. Table C-2 lists the mode switches and indicates which new switch replaces each.

Table C-2. Replacements for Mode Switches

Mode Switch	New Switch
BLANKED	BLANKED_VISUAL display-type visual switch
BLINKING	BLINKING_VISUAL display-type visual switch
INVERSE	INVERSE_VISUAL display-type visual switch
UNDERLINED	UNDERLINED_VISUAL display-type visual switch
LOW_INTENSITY	LOW_INTENSITY_VISUAL display-type visual switch
HIGH_INTENSITY	HIGH_INTENSITY_VISUAL display-type visual switch
INPUT_DISABLED	INPUT_FIELD data-state switch
NO_OVERLAY	FORCE_OVERLAY_MODE and FORCE_INSERT_MODE display-type action switches
AUTO_TAB	AUTO_TAB_TO_NEXT_FIELD display-type action switch
IMMEDIATE_RETURN	TRAP_ON_FIELD_EXIT display-type action switch
NOT_EDITABLE	DISAPPEARING_DEFAULT data-state switch
TRAP	TRAP_ON_FIELD_ENTRY display-type action switch

The polarity of some switches has been reversed. For example, the INPUT_FIELD data-state switch is the inverse of the DISABLE_INPUT mode switch. Note also that the initial editing mode formerly set by the NO_OVERLAY mode switch is handled somewhat differently by the display-type options. For more information, see the descriptions of the specific switches in Chapter 8, "Data States," and Chapter 7, "Display Types."

Dynamically Altering Forms

With the old FMS run-time software, usually one accept had to perform **all** the operations on a particular form: not only initializing and displaying the form, but also making various alterations to it. With the new FMS run-time software, you can separate the different operations to make the program easier to read and maintain. Usually, it is easier to use a perform screen update statement to alter a form between displays (or between initialization and the first display), rather than adding display-type clauses and field descriptions to the perform screen initialization, perform screen input, and perform screen output statements.

For example, in the following fragment, a field description is used in the accept statement to change one field of the form to low intensity on initial display. The modes table is subsequently updated to change the field back to normal intensity on redisplay.

```
move 0 to redisplay_switch.

perform old-intensity until ((key_code equal CANCEL)
                             or (code not equal 0)).

old-intensity.

    accept form (employee_info) update (employee_fields)
        keyused (key_code) redisplay (redisplay_switch)
        modes (modes_table) status (code),
        field (PHONE_ID) mode (LOW_INTENSITY).
    .
    .
    .

if redisplay_switch not equal 0 then
    move 1 to redisplay_switch
    move modes_info(PHONE_ID) - LOW_INTENSITY to modes_info(PHONE_ID).
```

You can rewrite this program to use the screen statements and display types. After initializing the form, you can use a perform screen update statement to create a new display type that includes the low-intensity attribute. You can then assign this display type to the phone field for the initial display of the form. After the initial display, you can reassign the original display type to the phone field. The following example implements these changes.

```
%replace LOW_INTENSITY_DT      by 11
%replace LOW_INTENSITY_VISUAL  by 16

01 display_types_table.
    02 display_types_info      comp-4
        occurs EMPLOYEE_INFO_MAX_IDS times.

01 old_display_type_id          comp-4.
.
.
.

procedure division.

* Initialize the form.

perform screen initialization form (employee_info) formid (emp_info_id)
    displaytypes (display_types_table) status (error_code).

if error_code not equal 0 then
    go to fatal-error.
```

(Continued on next page)

(Continued)

- * Define a display type with the low-intensity attribute.

```
perform screen update status (error_code),  
    displaytype (LOW_INTENSITY_DT) visual (LOW_INTENSITY_VISUAL).
```

```
if error_code not equal 0 then  
    go to fatal-error.
```

- * Save the current display-type ID for the phone field. Assign
- * the ID of the new display type to the phone field to make it
- * low intensity on initial display.

```
move display_types_info(PHONE_ID) to old_display_type_id.  
move LOW_INTENSITY_DT to display_types_info(PHONE_ID).
```

```
perform reassign until ((key_code equal CANCEL)  
    or (error_code not equal 0)).
```

```
reassign.
```

```
perform screen input formid (emp_info_id) update (employee_fields)  
    displaytypes (display_types_table) status (error_code).
```

```
.  
.  
.
```

- * Reassign the original display-type ID to the phone field.

```
move old_display_type_id to display_types_info(PHONE_ID).
```


In the following fragment, the phone field is changed to output-only on initial display and then changed back to an input field on redisplay.

```

%replace CANCEL          by -1
%replace INPUT_DISABLED  by 256
.
.
.

procedure division.

move 0 to redisplay_switch.

perform old-display until ((key_code equal CANCEL)
                        or (error_code not equal 0)).

old-display.

    accept form (employee_info) update (employee_fields) keyused (key_code)
    redisplay (redisplay_switch) modes (modes_table) status (error_code),
    field (PHONE_ID) mode (INPUT_DISABLED).

    if error_code not equal 0 then
        go to fatal-error.
        .
        .
        .

    if redisplay_switch not equal 0 then
        move modes_info(PHONE_ID) - INPUT_DISABLED to modes_info(PHONE_ID)
        move 1 to redisplay_switch.
        .
        .
        .

```

Because the INPUT_DISABLED mode switch has been replaced by the INPUT_FIELD data-state switch, use the datastates form option to convert this example. The converted code is as follows:

```
%replace CANCEL          by -1
%replace COPY_DATASTATE  by 256
%replace INPUT_FIELD     by 32

01 data_states_table.
    02 data_states_info    comp-4 occurs EMPLOYEE_INFO_MAX_IDS times.
        .
        .
        .

procedure division.

perform screen initialization form (employee_info) formid (emp_info_id)
    datastates (data_states_table) status (error_code).

if error_code not equal 0 then
    go to fatal-error.

* Make the phone field output-only.

move data_states_info(PHONE_ID) - INPUT_FIELD to data_states_info(PHONE_ID).

perform new-disable until ((key_code equal CANCEL)
    or (error_code not equal 0)).

new-disable.

    perform screen input formid (emp_info_id) update (employee_fields)
        datastates (data_states_table) options (COPY_DATASTATES)
        keyused (key_code) status (error_code).
        .
        .
        .

* Make the phone field an input field.

move data_states_info(PHONE_ID) + INPUT_FIELD to data_states_info(PHONE_ID).
    .
    .
    .
```

Note that the datastates form option is output-only in the perform screen input statement unless you set the COPY_DATASTATES options form option switch to true. For information on the options form option, see Chapter 5, "Form Options."

Output-Only Forms

If the form is output-only, you can replace the accept statement with a perform screen initialization statement and a perform screen output statement.

Multiple Forms

If the application might display more than one form in a particular window, you must either execute a perform screen initialization statement every time a different form is displayed (that is, every time an initial display of the form is required), or initialize each form once, and cache the forms that are not being displayed. Caching forms is faster than reinitializing. For information on caching forms, see Chapter 12, "Form Caching."

Binding the New Application

After you have compiled the revised program and created a new form object module with the new Forms Editor, you are ready to bind the new application. The run-time routines that support the new FMS functionality are located in the directory (master_disk)>system>icss_fms_object_library. You must include this directory in your object library search paths when you bind the application. For example, the following command binds an application in your current directory:

```
bind program_name -search (master_disk)>system>icss_fms_object_library
```

This command works if *program_name* is the name of the object module for your program, and all form object modules that the program references are within your normal object module search paths.

Appendix D:

Terminal Requirements

This appendix describes the features a terminal type must include to support FMS applications and to support the Forms Editor.

Input Requests

This section lists the terminal-type input requests used by the Forms Editor and by FMS applications.

The Forms Editor

Menu Edit Requests. To enable the Forms Editor request menu, you must define a sequence for the menu input request. Alternately, you can choose not to use a request menu and instead enable each of the menu requests as direct edit requests. To do this, define sequences for each of the input requests listed in Table D-1.

Table D-1. Alternatives for Menu Input Requests

Input Request	Purpose
add/modify-field	Alternative for (MENU) F request
define/modify-video-attributes	Alternative for (MENU) V request
delete-field	Alternative for (MENU) D request
en/disable-line-number-mode	Alternative for (MENU) N request
en/disable-overlay-mode	Alternative for (MENU) O request
en/disable-request-menu-display	Alternative for (MENU) A request
enter-field	Alternative for (MENU) E request
global-replace	Alternative for (MENU) G request
insert-window-field	Alternative for (MENU) I request
insert-literal	Alternative for (MENU) L request
quit	Alternative for (MENU) Q request
read-field	Alternative for (MENU) R request
set-bell-column	Alternative for (MENU) Z request
set/modify-form-attributes	Alternative for (MENU) S request
show-exact-form	Alternative for (MENU) X request
update-fields	Alternative for (MENU) U request
write	Alternative for (MENU) W request

Direct Edit Requests. To enable the direct edit requests listed in Chapter 4, “The Forms Editor,” you must define sequences for the following input requests.

back-space	enter	redisplay
back-tab	goto,beginning	return
blanks,left	goto,column	right
blanks,right	goto,down	save
cancel	goto,end	scroll,down
change-case,down	goto,line	scroll,left
change-case,up	goto,mark	scroll,multiple-down
column	goto,up	scroll,multiple-left
cycle	help	scroll,multiple-right
cycle-back	insert-default	scroll,multiple-up
del	insert-saved	scroll,right
delete	interrupt	scroll,up
delete,blanks	left	up
delete,left	line-feed	word,change-case,down
delete,return	mark	word,change-case,left
delete,right	menu	word,change-case,up
delete,word	next-screen	word,left
discard	previous-screen	word,right
down		

FMS Applications

This section lists the generic input requests that establish actions that a user can execute from within an FMS form.

The following requests allow the user to edit a field value. These sequences can also be used to edit a command line.

back-space	goto,beginning	word,change-case,left
del	goto,end	word,change-case,up
delete,left	left	word,left
delete,right	right	word,right
delete,word	word,change-case,down	

In addition to these requests, the requests in Table D-2 have specific uses in FMS forms.

Table D-2. Forms-Related Generic Input Requests

Input Request	Purpose
back-tab	Move cursor to previous field.
cancel	Cancel form.
cancel-form	Cancel form.
cycle	Display next cycle value.
cycle-back	Display previous cycle value.
display-form	Refresh all currently displayed forms.
down	Move cursor down to a new field.
en/disable-overlay-mode	Change edit mode of current field.
enter	Submit the current form.
erase-field	Set current field to its null value.
function-key-0	Submit the current form.
function-key-k	†
help	Display help text for current field.
insert-default	Set current field to its initial value.
insert-saved	Set current field to its initial value.
line-feed	Set current field to its null value.
redisplay	Redisplay the current field value.
return	Move cursor to next field.
tab	Move cursor to next field.
up	Move cursor up to a new field.

† You can set the meaning of each of the generic function key sequences with the maskkeys form option.

At minimum, a terminal type to be used for FMS applications should define sequences for the following input requests:

- left
- right
- back-space or del
- return or tab
- enter
- cancel or cancel-form

These requests allow the user to edit field values, move from field to field, and submit or cancel the form. Note that in a cycle field, the `right` and `left` requests have the same meaning as the `cycle` and `cycle-back` requests, respectively.

Output Requests

This section lists the terminal-type output requests used by the Forms Editor and by FMS applications.

The Forms Editor

The Forms Editor requires that sequences be defined for the following output requests:

- beep
- clear-to-end-of-line
- clear-to-end-of-screen
- display-block
- enter-graphics-mode
- half-intensity-off
- half-intensity-on
- leave-graphics-mode
- position-cursor
- set-attributes

Specific line-graphics code points can be defined instead of the `enter-graphics-mode` and `leave-graphics-mode` sequences.

For the most efficient operation, the following output sequences should also be defined:

- delete-lines
- goto-page
- insert-lines
- scroll-down
- scroll-up
- set-scrolling-region

If the terminal does not support the `set-scrolling-region` request, you can define the `freeze-lines` and `unfreeze-lines` requests instead.

FMS Applications

In order to run an FMS application, a terminal must define sequences for the following output requests:

```
position-cursor
clear-screen (or clear-to-end-of-screen)
```

With only these requests defined, forms can be displayed, but many FMS features are not available. For full FMS functionality and efficiency, the following requests should also be defined:

```
beep
clear-to-end-of-line
clear-to-end-of-screen
cursor-off
cursor-on
delete-chars
end-25th-line
enter-insert-mode
insert-chars
leave-insert-mode
reset-25th-line
set-attributes
set-cursor-blinking-block
set-cursor-blinking-underline
set-cursor-format
set-cursor-invisible
set-cursor-steady-block
set-cursor-steady-underline
start-25th-line
```

Attribute Requests

To take full advantage of FMS attribute features, a terminal type must define a sequence for each of the 64 terminal-type attribute requests. If the terminal does not support a particular attribute, replace it with a suitable alternative. For example, if a terminal does not support the high-intensity attribute, you might replace that attribute with the normal-intensity attribute.

)

)

)

)

)

Appendix E:

Form Storage Sizes

For asynchronous terminals, the total storage size limit for a form is four pages (16,234 bytes) of memory. For other device types, the storage size limit is determined by the available heap space.

Table E-1 shows how much storage space is required for various form components.

Table E-1. Storage Requirements of Form Components

Component	Bytes Used
Form header	184 bytes
Each field	52 bytes
Each input field	twice the field length in bytes
Each output field	field length in bytes
Each background text string	52 bytes
Each display type	26 bytes
Each picture	length of picture in bytes
Each help string	length of string in bytes
Each cycle value	4 bytes, plus the length of the value in bytes

In addition to the storage requirements listed in Table E-1, the user heap contains some additional data structures for each form and additional arrays for each port used.

If you must reduce the storage size of a form, the most effective method is usually to reduce the number of fields.

Appendix F:

Global Control Operations

This appendix discusses some global control operations that affect FMS forms. These operations are performed by calls to the `s$control` subroutine.

Within a program, you can invoke `s$control` as follows:

```
01 port_id          comp-4.  
01 opcode           comp-4.  
01 control_info     data_type.  
01 error_code       comp-4.
```

```
call 's$control' using port_id, opcode, control_info, error_code.
```

The first two arguments of `s$control` are input arguments, the third is an input-output argument, and the last is an output argument. The actual type of control information passed in the third argument depends on the opcode value given in the second argument.

For complete information on `s$control`, see the *VOS Communications Software: Asynchronous Communications (R025)*.

Forms Input Mode

To use forms input mode through the terminal port, you can use the `s$begin_forms_input` and `s$end_forms_input` subroutines as described in Chapter 14, "Subroutines." However, if you want to use forms input mode through another port, you must call `s$control` with one of the following opcodes.

- SET_INFO_OPCODE (202)
- SET_MODES_OPCODE (207)

Every port has an associated terminal information data structure that includes a set of 32 switches that determine the port modes. The `SET_MODES_OPCODE` allows you to establish new values for these modes. The `SET_INFO_OPCODE` allows you to establish new values for the modes and for other values within the terminal information structure.

The terminal information record is as follows:

```

01 terminal_info.
    02 version                comp-4.
    02 modes                  comp-5.
    02 line_length            comp-4.
    02 screen_size            comp-4.
    02 pause_lines            comp-4.
    02 prompt_chars            picture x(8) display-2.
    02 continue_chars          picture x(8) display-2.
    02 pause_chars             picture x(20) display-2.
    02 escape_char             picture x display.
    02 line_state              picture x display.
    02 obs_erase_char          picture x display.
    02 flow_on_char            picture x display.
    02 flow_off_char           picture x display.
    02 cursor_format           comp-4.
    02 flags                   comp-4.
    02 current_input_section    comp-4.
    02 filler                  comp-4.
    02 terminal_type_name       picture x(32) display-2.
    02 tabs-table.
        03 tabs comp-4 occurs 26 times.
    02 event_id                comp-5.
    02 terminal_type_number     comp-4.
    02 baud_rate               comp-4.
    02 process_id              comp-5.
    02 channel_name             picture x(32) display-2.

```

You can define constants for the mode switches as follows:

```

%replace  OS_FUNCTION_KEY_INPUT      by 2097152
%replace  OS_BREAK_TABLE_RECORD      by 1048576
%replace  OS_INTERRUPT_KEY_ENABLED    by 524288
%replace  OS_FORMS_INPUT              by 262144
%replace  OS_COMPLETE_WRITE           by 131072
%replace  OS_BULK_RAW_INPUT           by 32768
%replace  OS_SMOOTH_SCROLL            by 16384
%replace  OS_GENERIC_INPUT            by 8192
%replace  OS_BLACK_ON_WHITE           by 4096
%replace  OS_KEY_CLICK_ON             by 2048
%replace  OS_DISPLAY_ENABLE           by 512
%replace  OS_BREAK_ENABLED            by 256
%replace  OS_EDITED_OUTPUT            by 128
%replace  OS_RAW_INPUT                by 64
%replace  OS_BREAK_CHAR               by 32
%replace  OS_DSL_FLOW                 by 16
%replace  OS_USE_BREAK_TABLE          by 4
%replace  OS_OUTPUT_FLOW              by 2

```

Note that the `OS_FORMS_INPUT` switch has the value 262144. If this switch is true, the port is in forms input mode. If the switch is false, the port is not in forms input mode.

If you use `SET_INFO_OPCODE`, you must pass the entire terminal information structure in the third argument of `s$control`. The value of version must be 1. If you use `SET_MODES_OPCODE`, the third argument to `s$control` must be a comp-5 value.

Typically, before calling `s$control` with either of these opcodes, an application calls `s$control` with the `GET_INFO_OPCODE` (201). This call returns the current terminal information structure for a port. The program can then alter this structure and use it in a call to `s$control` with the `SET_INFO_OPCODE`; or the program can alter only the modes and pass the new modes in a call to `s$control` with the `SET_MODES_OPCODE` as in the following fragment.

```
%replace OS_FORMS_INPUT          by 262144

01 opcode      comp-4.
01 new_modes   comp-5.
.
.
.

move GET_INFO_OPCODE to opcode.

call 's$control' using port_id, opcode, terminal_info, error_code.

if error_code not equal 0 then
    go to fatal-error.

move modes of terminal_info to new_modes.

move SET_MODES_OPCODE to opcode.

call 's$control' using port_id, opcode, new_modes, error_code.

if error_code not equal 0
    go to fatal-error.
.
.
.
```

For more information on the `SET_INFO_OPCODE`, `SET_MODES_OPCODE`, or `GET_INFO_OPCODE`, see the *VOS Communications Software: Asynchronous Communications (R025)*.

Knocking Down Forms

An application can call `s$control` to knock down a form that is being displayed by another process or another task. To *knock down* a form means to cause the form to be submitted immediately, without user action. After the form is knocked down, the port can be used to display another form or to perform other I/O.

When a form is knocked down, the current field values are stored into the field-value variables **without** any validation. If a program or task that displays a form uses the `keyused` option, the value `-6` is returned in that option.

You can use either of two `s$control` opcodes to knock down a form.

- `KNOCK_DOWN_FORM_OPCODE` (240)
- `KNOCK_DOWN_FORM_OK_OPCODE` (264)

These opcodes are the same, except that if you use `KNOCK_DOWN_FORM_OK_OPCODE` and there is no form to knock down, `s$control` returns a nonzero error code, `e$invalid_form_id` (3794).

The code that displays a form and the code that knocks down that form must be coordinated to properly handle this action. Usually, forms are knocked down only by another task within the same application, rather than by another application.

Appendix G:

Stratus Character Code Set

Table G-1. Stratus Character Code Set

Hex Digits 1st 2nd	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0			SP	0	@	P	`	p			NBSP	°	À	Ð	à	ð
1			!	1	A	Q	a	q			í	±	Á	Ñ	á	ñ
2			"	2	B	R	b	r			¢	²	Â	Ó	â	ò
3			#	3	C	S	c	s			£	³	Ã	Ô	ã	ó
4			\$	4	D	T	d	t			□	'	Ä	Ö	ä	ö
5			%	5	E	U	e	u			¥	μ	Å	Õ	å	ø
6			&	6	F	V	f	v			¦	¶	Æ	Ö	æ	ö
7				7	G	W	g	w			§	•	Ç	X	ç	+
8			(8	H	X	h	x			"	,	È	Ø	è	ø
9)	9	I	Y	i	y			©	¹	É	Ù	é	ù
A			*	:	J	Z	j	z			ª	º	Ê	Ú	ê	ú
B			+	;	K	[k	{			«	»	Ë	Û	ë	û
C			,	<	L	\	l				¬	¼		Ü	ü	ü
D			-	=	M]	m	}			SHY	½		Ý	ý	ý
E			.	>	N	^	n	~			®	¾		Þ	þ	þ
F			/	?	O	_	o				—	¿		ß	ÿ	ÿ



Control characters that do not have a graphic representation in ISO 8859/1

PD0005

The left side of the table is the ASCII character set. The right side of the table is the Latin alphabet No. 1 character set.

Glossary

access

To read from or write to a file or device. See also **access mode**, **access right**, and **access type**.

access mode

The method that the I/O system uses to access records for reading or writing. The access modes are sequential, random, and indexed.

access right

A designation that determines the operations a user is permitted to perform on a file or directory. Access rights to a file are null, execute, read, and write. Access rights to a directory are null, status, and modify.

access type

A type of I/O operation performed on a file or device. Access types are input, output, append, update, dirty read, and dirty notify.

action switches

In FMS, a series of 32 switches associated with a display type that define the behavior of fields having that display type.

address

The location of an area of storage. An address is a four-byte value.

address space

See **virtual address space**.

alphanumeric field

In FMS, a field that can contain both alphabetic and numeric characters. See also **numeric field**.

American Standard Code for Information Interchange (ASCII)

A standard 7-bit character representation code that VOS stores in an 8-bit byte.

argument

A character string that specifies how a command, request, subroutine, or function is to be executed.

array

An ordered set of program objects all of the same type.

array element

An individual object in an array.

array field

In FMS, a field that consists of a set of simple fields having sequentially numbered IDs.

ASCII

See **American Standard Code for Information Interchange**.

asynchronous communications

Data transmission where the time interval between the transmission of characters can vary from one character to the next. The beginning and end of each character transmitted is determined by means of start and stop bits preceding and following each character.

attribute

In FMS, a characteristic of a field, especially the visual characteristics: high intensity, low intensity, blinking, blanked, inverse video, and underlined.

background text

Text within a form, but not in a field. Background text includes titles and labels that identify fields.

backup

In Emacs and in the VOS Word Processor, the online saving of an unaltered original copy of a file while you are making and saving editorial changes to that file. Using the `-backup` argument of the `emacs` command or of the `edit` command causes VOS to create a backup copy of an existing file the first time during a session that you write the contents of a buffer to that file.

bind

To combine a set of one or more independently compiled object modules into a program module. Binding compacts the code and resolves symbolic references to external programs and variables that are shared by object modules in the set and in the object library. See also **library**.

binder

The program that combines a set of independently compiled object modules into a program module. The binder is invoked with the bind command.

Boolean value

A bit string of length one indicating a truth value. The value '1'b means true, and '0'b means false.

break

1. A signal (or to send a signal) that interrupts a program being executed and places the process executing the program at break level.
2. To bring the communications line temporarily to a low-voltage condition in order to request attention from the running program or the operating system.

buffer

For a terminal, a temporary data storage location in the terminal's memory. The buffer can be used to compensate for differences in transmission rates or temporarily store characters until the computer can accept them.

built-in functions

Functions that are predefined within the VOS COBOL language.

byte

1. Eight bits of data. An unsigned byte variable can contain integer values in the range 0 to 255; a signed byte variable can contain integer values in the range -128 to 127.
2. The unit of storage consisting of eight contiguous bits.

cache

To store a form for later retrieval. For example, an FMS application can cache the internal representation of an FMS form that is not currently displayed on the screen.

character

1. A symbol, such as a letter of the alphabet or a numeral, or a control signal, such as a carriage return or a backspace. Characters are represented in electronic media by character codes.
2. A character code.

character code

1. A numeric value used to represent a character according to a specified system. For example: *The ASCII (character) code for A is 41 (hexadecimal)*. In the Stratus Internal Character Coding System, the bit representation of a character code can occupy one or two bytes, depending on the character set.
2. A particular system used to assign numeric values to characters. For example, *the American Standard Code for Information Interchange (ASCII)*.

character string

An ordered set of characters from one or more character sets. The length of a character string depends on the size of each character (one or more bytes, depending on the character set), the number of characters in the string, and the use of single-shift and locking-shift characters within the string. Character strings are evaluated from left to right.

command

A program invoked from command level, either interactively or as a statement in a command macro.

command line

A set of one or more commands, separated by semicolons. Pressing one of the following keys terminates a command line: **RETURN**, **ENTER**, **DISPLAY FORM** (and, on V101 terminals only, **EGI**, **ES**, **EM**).

compiler

A program that translates a source module (source code) into machine code. The generated machine code is stored in an object module.

condition

An exceptional occurrence during the execution of a program. In VOS PL/I, a program can be set up to detect a condition by using the **on** statement. Execution of an **on** statement establishes an on-unit as a routine to be executed when a specified condition occurs. The other VOS languages use service subroutines, such as **s\$enable_condition**, to work with conditions.

condition handler

A routine, defined by a program, that is invoked in response to a condition signaled during the execution of the program. In VOS PL/I, an on-unit.

constant

In a program, a value that cannot be changed.

continuous forms mode

See **forms input mode**.

conversion

The process of transforming a value from one data type to another.

current directory

The directory currently associated with your process. VOS uses your current directory as the default directory when you do not specifically name the directory containing an object that you want VOS to find. For example, if you supply a relative path name in a command, VOS uses the current directory as the reference point from which to locate the object in the directory hierarchy.

When you log in, your current directory is set to your home directory. You can change the directory that is your current directory with the `change_current_dir` command or the `s$change_current_dir` subroutine.

current form

The form in memory that was most recently displayed or initialized in a specific window. Each window can have only one current form.

current word

In an editor, the word or portion of a word to which the cursor is currently positioned; the text that is affected by certain word edit requests.

cursor

A marker on a screen showing where the next character should appear.

cycle field

In FMS, a field that has a specific list of possible values. The user cannot type in a cycle field but can change the displayed value by using the **CYCLE** key and other keys.

cycle list

The list of possible values for a cycle field.

data state

In FMS, a series of 16 switches associated with a field that describe the current state of the data in that field.

data type

The collective attributes of a value, variable, or object that determine the operations that can be performed on that value and the scheme by which the value is stored.

default

The value or attribute used when a necessary value or attribute is omitted.

default character set

In ICSS strings or in text files, the supplementary graphic character set that, in the absence of single- or locking-shift characters, is represented by the character codes in the range A0-FF (hexadecimal). The default character set for a file can be set by the `create_file`, `set_text_file`, or `emacs` command. Some of the ICSS built-in functions in VOS programming languages permit specification of the default character set as an argument. If a default character set is not specified, the default is usually assumed to be Latin alphabet No. 1.

default value

The value that VOS uses if a specific value is not supplied.

device

Any hardware component that can be referenced and used by the system or users of the system and that is defined in the device configuration table. Terminals, printers, tape drives, and communications lines are devices.

direct edit request

In an editor, a request issued by a user from the edit buffer, rather than from a request menu. See also **menu edit request**.

display list

In FMS, a form stored in memory by a program. The display list is the version of a form that a program can manipulate.

display type

A specific set of field characteristics that can be applied to one or more fields in one or more forms. A field's display type describes the field's appearance, behavior, and certain restrictions on field values. Some display types are predefined in the Forms Editor; others can be created dynamically.

edit buffer

The location in which material being edited is stored during an edit session.

edit request

A request issued by a user from within an editor. An edit request can be either a **direct edit request** or a **menu edit request**.

editing session

In the VOS Word Processor, the uninterrupted period of time between when you start using the text editing program and when you stop. There is no limit to the duration of an editing session.

editor

A program used to create and modify certain types of files.

For text files, Stratus provides two screen editors (designed for video display terminals) and one line editor (designed for printing terminals). The screen editors are called by entering `emacs` or `edit`, and the Line Editor is called by entering `line_edit`.

For FMS forms, Stratus provides the Forms Editor.

error code

An arithmetic value (usually a two-byte integer) indicating what, if any, error has occurred; usually, a VOS status code. An error code argument is often included in subroutines. See also **status code**.

expression

A series of one or more operands and, usually, one or more operators that yield a value. An identifier without an operator is an expression that yields a value directly.

extents

The bounds specification of an array or the length specification of a string value.

false

Unambiguously incorrect; the opposite of true; evaluating to a bit-string value of '0'b. See also **true**.

field

An area of a screen form in which values can be entered or displayed.

field definition file

A file produced by the Forms Editor containing the definition of a field. The field definition can be read from the file by the Forms Editor later in that edit session or in a subsequent session.

field description

A clause in an accept or screen statement that defines or modifies a field in a form.

field edit mode

In FMS, the mode that determines how new characters are added to a field. The field edit mode can be either insert or overlay. See also **insert mode** and **overlay mode**.

field ID

An integer that identifies a field in a form. Each field in a form has an ID that is unique within that form.

field-IDs file

An include file produced by the Forms Editor containing definitions of numeric constants for the field IDs of a form.

field modes

In old-style FMS, a series of 16 switches that describe characteristics of a field. In new FMS, field modes have been replaced by display types and data states.

field picture

A string of special characters that restricts the valid characters for each position within the field. A field picture is the same length as the field it describes. Each character in the field picture indicates which characters are valid in the corresponding position within the field.

field-value variable

The program variable into which the Forms Processor returns the value of a field.

field-values file

An include file produced by the Forms Editor containing declarations for field-value variables.

file

A set of records or bytes stored on disk or tape as a unit. A disk file has a path name that identifies it as a unique entity in the system's directory hierarchy. Attributes of a disk file, such as its size and when it was created, are maintained in the directory containing the file.

filtering

The process of removing certain characters from a numeric value before displaying that value or before storing that value.

form

See **screen form**.

form definition file

A file produced by the Forms Editor containing a description of a form that can be read by the Forms Editor during a subsequent invocation. A form definition file always has the suffix `.form`.

form ID

A two-byte integer that identifies a form within a program. A form ID is generated by the Forms Processor when a form is initialized. No two active forms within an application can have the same form ID.

form object module

A file produced by the Forms Editor containing a description of a form that can be bound into a program module and referenced by an application program.

Forms Editor

A program used to create and modify FMS screen forms that can later be referenced by application programs. A form created with the Forms Editor is called a predefined form.

forms input mode

A mode of asynchronous I/O in which all input to the application is entered through FMS forms.

Forms Processor

The systems run-time software that manages FMS forms. An application program invokes the Forms Processor by executing a screen or accept statement.

forms reference count

An internal integer value associated with a form. The Forms Processor uses the forms reference count to determine when the form can be discarded.

full path name

For a file, directory, or link, a name that is composed of the name of the system, the name of the disk, the names of the directories that contain the object, and the name of the file, directory, or link.

For a device, a name that is composed of the name of the system and the name of the device.

A full path name refers to only one object; an object has only one full path name. (However, many links can refer to the same object.)

function key

One of the keys on the keyboard that generally produces a nonprinting character or a sequence of nonprinting and printing characters. Examples include the numbered function keys (F0, etc.), the arrow keys, and the BACK SPACE key.

generic function keys

A set of 32 generic sequences recognized by the VOS communications software and assigned to a specific key or combination of keys within a terminal type definition.

graphic character

A symbol, such as a letter of the alphabet, a numeral, or a punctuation mark, as opposed to a control character. In the Stratus Internal Character Coding System, graphic characters are represented by codes in the ranges 21-7E and A0-FF (hexadecimal).

grouping character

A character used to divide the digits of a large number, usually into groups of 3. For example, in the value 1,000,000, the commas act as grouping characters.

heap

A collection of randomly accessible storage associated with a process and available for allocation. See also **user heap**.

help message

Explanatory text displayed to the user.

hexadecimal notation

Notation of numbers in base 16.

ICSS

See **International Character Set Support**.

incidental data

Data other than field values that is returned from a form to an application program. Incidental data includes information such as the key used to submit or cancel a form and the location of the cursor.

include file

A file that the compiler includes in the source module used by the compilation process. The name of the include file must be specified in a language-specific directive within the source module.

include library

A series of directories that VOS searches for include files.

initial display

A display of an FMS form in which the entire form, including all fields and background text, is output to the screen. See also **redisplay**.

initial output value

The value displayed in a field of an FMS form on initial display.

input argument

In VOS COBOL, an argument that supplies a value to a called procedure, but is itself never altered; input arguments can be passed either by-reference or by-value.

input field

A field in an FMS form to which the user can position the cursor and then usually type or cycle to a value. See also **output field**.

insert mode

A terminal mode in which characters written to the terminal are inserted at the current location. Existing characters that follow are pushed to the right, not overwritten. See also **overlay mode**.

intensity

The brightness with which characters are displayed on a terminal screen.

International Character Set Support (ICSS)

The ability of the operating system to represent text in languages other than English.

inverse video

A terminal mode in which the characters on the screen are black and the screen background is amber. Also called *black-on-white*.

I/O

Input and output.

justification

The positioning of a value within an FMS field. A value can be left-justified (begin at the left edge of the field), right-justified (end at the right edge of the field), or centered.

keystrokes file

A file that contains representations for keystrokes you make during an edit session, in sequence.

left graphic character

A character located in the range of 21x to 7Ex in the Stratus internal character coding system. Left graphic characters compose the ASCII character set.

library

One or more directories in which VOS looks for objects of a particular type. There are four kinds of libraries defined by VOS.

- Include libraries, in which the compilers search for include files
- Object libraries, in which the binder searches for object modules
- Command libraries, in which the command processor searches for commands
- Message libraries, in which the operating system searches for message files associated with individual .pm files

One of each of these libraries is available in the >system directory of each module for all processes running on the module. In addition, you can define your own libraries.

library paths

The set of directories associated with a particular library. Each process has one set each of include library paths, object library paths, command library paths, and message library paths.

locking-shift character

A user-transparent character in an array or string, indicating that the remaining characters in the key or record are from a character set other than the default character set.

masked key

A key (or combination of keys) that has been defined as an alternative **ENTER** or **CANCEL** key for an FMS form. Key combinations that are assigned to generic function key requests in the terminal type table can be masked.

master form

A form that is displayed in the master window rather than in a window field of another form. See also **subform**.

master window

A window that comprises the entire screen.

menu

In FMS, a form whose purpose is to present a selection of additional forms that you can display.

menu edit request

In an editor, a request that is issued by first invoking a request menu and then selecting a request from that menu. See also **direct edit request**.

mode switch

One of the switches that comprise the FMS field modes.

modify access

A type of directory access that means a user has full access to the contents of the directory, including the ability to create, delete, and rename objects.

module star name

A name that contains one or more asterisks or consists solely of an asterisk, used to specify a set of modules in a system. An asterisk can be in any position in the name, and each asterisk represents zero or more characters. A module star name can be used alone or as part of a full module path name; however, in a full module path name there can be no asterisks in the system name. When a module star name consists solely of an asterisk, it represents all modules in the current system. See also **star name**.

nested

Contained within another of its kind: a nested statement occurs within another statement; a nested block occurs within another block.

null field value

The value displayed in a field when it is empty. If the field does not have a picture, the null field value is a string of spaces. If the field has a picture, the null field value might include self-insertion characters and other characters.

numeric field

In FMS, a field that can contain only numeric values. See also **alphanumeric field**.

object library

A series of directories that VOS searches for object modules.

object module

A file produced by a compiler, containing the machine code version of one or more procedures; it usually contains symbolic references to external variables and programs. To execute the program, an object module must be processed by the binder to produce a program module and then loaded by the loader.

opcode

In VOS, an operation code; a two-byte integer value passed to `s$control`. Each opcode directs `s$control` to execute a different device control operation.

operand

A variable, constant, expression, or value upon which an operator or statement works.

optional argument

An argument for which the operating system does not need a value to execute the command.

output argument

A subroutine argument that can be altered by the called procedure. Output arguments must be passed by-reference.

output field

In FMS, a field in which the application can display values that the user cannot change. Some output fields are output-only fields which cannot be changed to input fields. Other output fields can be dynamically changed to input fields. The term **output field** sometimes refers only to the latter type. See also **input field** and **output-only field**.

output-only field

In FMS, an output field that cannot be dynamically changed to an input field.

output-only form

In FMS, a form that does not contain any input fields and does not have any keys masked. This term can also refer to any form displayed with the `output screen` statement.

overlay mode

A terminal mode in which each character entered at the keyboard replaces the character at the current cursor position. See also **insert mode**.

parameter

A value upon which a procedure operates. The actual value is supplied when the procedure is called.

path name

A unique name that identifies a device or locates an object in the directory hierarchy. See also **full path name** and **relative path name**.

picture

A string of picture characters used to describe pictured data; also, a field picture.

picture characters

A set of characters used to describe pictured data or used in a field picture.

pictured data

Program objects in COBOL or PL/I that are described pictorially by a string of picture characters.

port

A data structure, identified by a name or ID, that you attach to a file or device for the purpose of accessing the file or device. When an executing program refers to a file or device, VOS uses the port having the same name or ID.

A port is created when it is attached and destroyed when it is detached.

port ID

A two-byte integer used to identify a port.

predefined field

In FMS, a field defined with the Forms Editor within a predefined form.

predefined form

In FMS, a form defined with the Forms Editor.

printable characters

Those characters that can be printed by most standard printers and displayed on most standard terminals; ASCII characters with rank greater than 31 and less than 127.

privileged process

A process of a user who is logged in as privileged.

program

One or more procedures, from one or more source modules, that together perform a task.

range field

In FMS, a field for which a range restriction has been specified. Range restrictions can be specified with the Forms Editor or with the range form option of a screen or accept statement.

read access

A type of file access that means a user can read the file or execute it if it is executable, but cannot write it.

redisplay

In FMS, a display of a form in which only changed fields are written to the screen. The rest of the form is assumed to be on the screen. See also **initial display**.

relative path name

A name that identifies a device or an object in the directory hierarchy without specifying its full path name.

request

In an editor, a keystroke or combination of keystrokes that cause a specific action to be performed.

required argument

A command argument for which you must specify a value.

required field

In FMS, a field that must contain a non-null value when the form is submitted.

reverse video

A video attribute that reverses the color of the background and the characters on the screen. For example, if the screen displays light characters on a dark background, the reverse video attribute changes the display to dark characters on a light background.

right graphic character

A character located in the range of A0x to FFx in the Stratus internal character coding system. Right graphic characters compose the Latin alphabet No. 1, katakana, kanji, and hangul character sets.

run

To execute a program.

run time

The time when a program module is invoked and executed.

scalar

A single, one-dimensional value or object; not an array or structure, though possibly an array element or structure member.

scale

An attribute of arithmetic data: fixed-point or floating-point.

scaling factor

The number of digits in a fixed-point arithmetic value that appear to the right of the radix point; the number of fractional digits.

screen

The display area of a video display terminal.

screen form

A form displayed on a video display screen.

scrolling

The movement of data across the screen.

shift character

See **single-shift character** and **locking-shift character**.

shift mode

1. A text file attribute indicating the types of shift characters in a file. See also **single-shift character** and **locking-shift character**.
2. What form of shift characters are in use. The possible forms are none, single shift only, locking shift only, or either single or locking shifts.

simple field

In FMS, a single non-array field.

single-shift character

A user-transparent character in a key or record, indicating that the next character is from a character set other than the default.

source module

A text file (single source program) containing language statements, compile-time statements, and comments that can be compiled to produce an object module.

special numeric characters

In FMS, characters that are valid in prefiltered strings, but are not valid in filtered strings.

star name

A name that contains one or more asterisks or consists solely of an asterisk. A star name can be used to specify a set of objects. Star names function in the following manner.

- An asterisk can be in any position in a star name.
- In a path name, a star name can be in the final object name position only.
- When the operating system matches non-star names to a star name, each asterisk represents zero or more characters.
- No name can contain consecutive asterisks; there must always be an intervening character.

Some names with asterisks function differently; see also **module star name** and **user star name**.

statement

One of several programming constructs consisting of identifiers, operators, and separators. A statement specifies an action or actions to be executed by a program.

status access

A type of directory access that means a user can display information about the directory, but cannot modify the directory by creating, deleting, or renaming objects.

status code

A two-byte integer, with an associated name, indicating the success, failure, or other status of an operation. A zero status code is generally associated with success.

VOS has four types of status codes.

- Error codes, whose names begin with the prefix **e\$**
- Message codes, whose names begin with the prefix **m\$**
- Query codes, whose names begin with the prefix **q\$**
- Response codes, whose names begin with the prefix **r\$**

status line

See **terminal status line** and **Stratus status line**.

Stratus status line

A line displayed in the terminal's bottom information line which provides information about the process using the terminal rather than about the terminal itself.

string

A character string or bit string. See also **character string**.

subform

In FMS, a form that is displayed in a window field within another form. See also **master form**.

subroutine

A sequence of statements that can be invoked as a set at one or more points in a program to execute a specific operation.

switch

A bit used to indicate a mode or state. A switch can be either true (on) or false (off).

symbol table

The part of a program module that contains data used for debugging the program. The symbol table allows the debugger to convert the names of user-defined variables to locations of data or instructions. For object modules, the symbol table is placed in the symtab region.

symtab region

The portion of an object module that contains the symbol table.

terminal

A device used for input and output. Most modern terminals have screens and keyboards; however, printing terminals (with keyboards) also fall into the class of terminal. If a terminal has its own hard-copy printer connected to it, that printer is considered part of the terminal.

terminal status line

A line displayed in one of the terminal's information lines that shows a number of the terminal's operating parameters.

terminal type

In VOS, a software, table-driven mapping facility that provides a certain degree of terminal independence to the Stratus asynchronous terminal software. Terminal types are typically defined for all different types of terminals that will be using the system. At times, terminal types are defined solely for the purpose of changing the input keystroke mappings. Terminal types are defined for the duration of a bootload by privileged processes executing the `define_terminal_type` command.

terminal type definition file

In VOS, a file that defines the relationship between generic input and output requests and terminal-specific input and output sequences. The terminal type definition file is the input that `define_terminal_type` uses to create the terminal type tables for a terminal type.

transaction

A sequence of operations that are performed as a unit. Typically, a transaction involves updating one or more pieces of data in a database.

trap

In FMS, an automatic form submission that occurs when the user positions the cursor into or out of specific fields.

trap field

A field for which a trap occurs. For a trap on field exit or trap on field entry, the trap field is the field on which the trap is set. For a vertical scroll trap, the trap field is the field from which the cursor is moved to cause the trap.

true

Unambiguously correct or valid; evaluating to the bit-string value '1'b. See also **false**.

truncate

To cut off; to remove excess digits without rounding.

uncommitted field

In FMS, a field which is defined as part of a form, but which does not have a defined position within the form.

user heap

Portion of a user's virtual address space in which VOS can allocate storage for the user's programs. The user heap is a free storage region located after the executable image in the user's region of virtual storage. It grows up towards the user stack, which grows down from the highest virtual address.

user star name

A user name containing one or two asterisks that is used to specify a set of users. When a user attempts to use a file or directory to which an access control list applies, the operating system checks the user's access by matching user star names on the list to actual users and groups.

Either component of a user star name (the person name or the group name) can be an asterisk, or both components can be asterisks. An asterisk as the first component matches all person names; an asterisk as the second component matches all group names. In arguments that accept user star names, if only a person name (or only a single asterisk) is given, the operating system appends .* to the name.

validation routine

In FMS, a procedure performed by an application program to validate the input value for a specific field. A validation routine is invoked as part of the validation suite for the field.

validation suite

In FMS, a series of checks and tests applied to an input field value when a form is submitted.

variable

A data item whose value may be changed by the execution of the program. In COBOL, a variable used in a numeric expression must be a numeric or a numeric-edited elementary item.

virtual address space

A set of addresses to which a process can refer. VOS gives each user a virtual address space of 4096 pages, of which the first 2048 pages belong to the kernel. Excluding the addresses used by VOS, the size of a user's virtual address space is either two megabytes or eight megabytes.

Virtual Operating System (VOS)

A Stratus operating system.

visual switches

In FMS, a series of 32 switches associated with a display type that describe the appearance of fields having that display type.

VOS

See **Virtual Operating System**.

window

In FMS, an area of a terminal screen in which a form can be displayed.

window field

In FMS, a field that defines a window in which a subform can be displayed.

write access

A type of file access that means a user can execute, read, and write the file.

zero suppression

The masking of zeroes, especially of leading integral or insignificant trailing fractional zeroes. Suppressed zeroes are usually replaced by space characters, although asterisks are also used.

Index

Special Characters

,, 3-2, 9-2, 9-3
\$, 4-46, 9-5
*, 9-5, 9-6
+, 9-5
-, 3-3, 9-2, 9-3, 9-5, 9-6
., 3-2, 9-2, 9-3
/, 3-3, 9-2, 9-3, 9-5, 9-6
%replace compile-time statement, 4-58
␣ key, 4-13
␣ key, 4-13
␣ key, 4-13
␣ key, 4-13

Numbers

3270 terminals, 4-49, 5-14
9 picture character, 3-2, 3-3, 9-2, 9-3

A

A picture character, 3-3, 9-2
accept_field_definitions directory, 4-6, 4-22, 4-23
accept statement, 1-3, 16-2, A-1
 converting to screen statements, C-2
 display list, A-2
 editing modes, A-6
 field justification, A-9
 field modes, A-5
 initial display, A-2
 initial output values, A-10
 old-style, A-1, A-2
 redisplay, A-2
Action attributes, 2-40, 7-11
 auto-tab, 2-13, 4-39, 7-13
 bank-teller decimal, 4-39, 4-46, 7-14
 force insert mode, 4-41, 7-14
 force overlay mode, 4-41, 7-14
 indexed cycle list, 4-40, 7-9, 7-15

 trap on field entry, 4-40, 7-14
 trap on field exit, 4-40, 7-13
action display-type option, 7-5, 7-11
Active forms, 3-9, 3-10
Add/modify field request, 2-10, 4-22, 4-25
 form, 2-10, 4-28
 old-style, B-7
alloc_screen_displaytype function, 15-4
alloc_screen_field function, 15-4
␣ key, 4-12
Alphanumeric fields, 3-3
Alphanumeric pictures, 3-3
ALTERABLE BY ACCEPT form option, 4-47
Application development, 2-1
Arrangement of fields, 2-3
array field option, 6-9
Array fields, 4-27, 4-30, 6-9
 spacing between elements, 4-30, 6-9
ARRAY LAYOUT field option, 4-27, 4-30
Arrow keys, 2-15, 4-13
ASCII character set, 3-6
Asterisk (*), 9-5, 9-6
Attributes, 2-21
 of background text, 2-23, 4-24, 4-50, 4-51
 of fields, 1-3, 2-14, 2-15, 2-22, 4-38
 of required fields, 2-22, 4-50
AUTO TAB display-type option, 2-13, 4-39
AUTO_TAB mode switch, A-5
AUTO_TAB_TO_NEXT_FIELD display-type switch, 7-13

B

B picture character, 3-3, 9-2, 9-3
␣ request, 4-13
␣ request, 4-13
BACKGROUND MODE form option, 2-23, 4-50
Background text, 1-1, 2-2, 2-9, 4-11
 dynamically defined, 6-2
 incidental data and, 2-5
 supplemental characters, 4-23
 video attributes, 2-23, 4-32, 4-50

BANK TELLER DECIMAL display-type option, 4-39
 BANK_TELLER_DECIMAL display-type switch, 7-14
 BANK TELLER DECIMAL form option, 4-46
 BASIC field option, 4-33
 BASIC form option, 4-51
 BEEP form option, 4-48
 beep form option, 5-2, 14-2
 Bell. *See also* beep form option
 setting column, 4-25
 Binding, 1-4, 2-42, C-11
 BLANKED mode switch, A-5
 BLANKED_VISUAL display-type switch, 7-21
 Blanking
 background text, 2-23, 4-24, 4-50, 4-51
 fields, 2-22, 4-38, 7-21
 required fields, 2-22, 4-50
 (BLANKS) (←) request, 4-13
 (BLANKS) (→) request, 4-14
 Blinking
 background text, 2-23, 4-24, 4-50, 4-51
 fields, 2-22, 4-38, 7-21
 required fields, 2-22, 4-50
 BLINKING mode switch, A-5
 BLINKING_VISUAL display-type switch, 7-21
 Buffer, 4-10
 Built-in functions, 15-1
 alloc_screen_displaytype, 15-4
 alloc_screen_field, 15-4
 display types, 15-2
 fields, 15-3
 find_screen_field, 15-5
 first_changed_field, 15-5
 first_screen_displaytype, 15-6
 first_screen_field, 15-6
 last_screen_displaytype, 15-7
 last_screen_field, 15-7
 next_changed_field, 15-8
 next_screen_displaytype, 15-9
 next_screen_field, 15-10
 prev_screen_displaytype, 15-11
 prev_screen_field, 15-12
 screen_field_position, 15-13

C



C field option, 4-33
 C form option, 4-51
 Caching forms, 3-12, 12-1, 16-19

(CANCEL) request, 2-21, 3-14, 4-14
 Canceling a form, 1-2, 3-14
 CENTER_FIELD_DATA display-type switch, 7-22
 center field option, 6-10, A-9
 (CHANGE CASE) (↓) request, 4-14
 (CHANGE CASE) (↑) request, 4-14
 Changed fields, 8-7, 15-1
 CHAR SET display-type option, 4-39
 Character set IDs, 3-7
 Character sets, 3-6, 4-32, 4-39, 6-22, 6-23, 7-15
 charset display-type option, 3-7, 7-15
 CHECK_3270_FORMS_MODEL options switch, 5-14
 CHECK 3270 MODEL form option, 4-49
 clear form option, 5-3, C-4
 Clearing the screen, 4-46, 5-3
 COBOL field option, 4-33
 COBOL form option, 4-51
 (COLUMN) request, 4-14, 4-27, 4-42
 COLUMN SPACING field option, 4-30
 Comma (,), 3-2, 9-2, 9-3
 Communication errors, 10-2
 Compiling, 2-42
 after form modification, 2-24
 Conditions
 error, 10-1
 Continuous forms mode. *See* Forms input mode
 Control transfer, 2-4, 3-13
 Conversion
 case, 9-2
 field values, 10-4
 old forms, 4-2, C-1
 old programs, C-2
 COPY_DATASTATE options switch, 2-38, 5-4, 5-14, 8-5
 copy statement, 2-27, 4-58
 cr sign characters, 9-5
 Currency symbol, 9-5, 9-6
 CURRENCY SYMBOL form option, 4-46
 Current form, 12-1
 Current line, 4-10
 Current word, 4-10
 Cursor
 movement keys, 4-13
 wide, 4-49, 5-14
 CURSOR FIELD form option, 4-50
 Cursor location, 2-5
 next, 5-13
 obtaining, 5-7
 setting, 4-50, 5-17

- cycle_array display-type option, 7-11, 7-17
- (CYCLE BACK)** request, 2-15, 4-14
- cycle display-type option, 7-7, 7-16
- cycle field option, 6-11, A-10
- Cycle fields, 1-3, 4-37, 7-7
 - defining, 2-15
 - optional, 2-16
- Cycle list, 1-3, 7-7
 - altering, 2-16, 7-9
 - inquiring about, 7-11
 - specifying, 2-16, 4-37, 7-8
- (CYCLE)** request, 2-15, 4-14

D

- Data-entry loop, 2-31
- Data states, 1-3, 2-36, 5-3, 6-11, 8-1
 - altering, 2-36
 - changing, 8-5
 - default, 6-4
 - in Forms Editor, 8-2
 - reading, 8-4
 - switches, 8-1
 - table, 2-37
- Data types, 2-11, 4-33, 4-34
- datastate field option, 6-5, 6-6, 6-11, 8-2
- datastates form option, 2-36, 5-3, 6-5, 8-2
- db sign characters, 9-5
- decimal_is_comma form option, 4-46, 9-4
- decimal_is_period form option, 4-46, 9-4
- Decimal point character, 4-46, 9-3, 9-4
- Declarations
 - field-value variables, 2-26, 4-53
- Defaults
 - attributes, 2-14, 2-15, 4-38
 - character set, 3-7, 4-32, 6-22
 - data states, 6-4
 - data types, 4-34
 - disappearing, 4-32, 8-6
 - display types, 6-3
 - field modes, A-6
 - output values, 2-16
 - video attributes, 3-4
- Define/modify video display modes request, 2-23, 4-24, 4-51
- Defining a form
 - dynamically, 6-2
 - with Forms Editor, 2-8, 4-1

- (DEL)** request, 4-14
- (DELETE)**  request, 4-15
- (DELETE)**  request, 4-15
- (DELETE)** **(BLANKS)** request, 4-15
- Delete field request, 4-21
- (DELETE)** request, 4-15
- (DELETE)** **(RETURN)** request, 4-15
- (DELETE)** **(WORD)** request, 4-15
- Designing a form, 2-2, 4-2
- Direct edit requests, 4-13, D-2
- DISABLE_ENTIRE_FIELD data-state switch, 4-27, 8-6
- DISABLE field option, 4-31
- Disabling fields, 4-27, 4-31, 8-6
- DISAPPEARING_DEFAULT data-state switch, 8-6
- DISAPPEARING field option, 4-32
- (DISCARD)** request, 4-15
- Discarding forms, 3-12, 12-2, 16-8
- Display list, 1-7, 3-10
 - accept statement, A-2
 - altering, 3-11, 16-6, 16-21
 - discarding, 3-12, 12-2, 16-8
 - displaying, 3-10, 16-12, 16-17
 - initializing, 2-27, 3-10, 16-9
 - inquiring about, 3-11, 16-15
 - saving, 3-12, 12-1, 16-19
- Display-type classes, 7-3
- Display-type descriptions, 2-40, 7-2
- Display-type IDs, 2-40, 7-1
 - maximum, 5-11
 - table, 2-41, 5-5
- Display-type options, 4-36, 7-2, 7-11
 - action, 7-5, 7-11
 - charset, 3-7, 7-15
 - cycle, 7-7, 7-16
 - cycle_array, 7-11, 7-17
 - picture, 7-18
 - range, 7-19
 - validate, 7-19
 - visual, 7-5, 7-20
- Display types, 1-3, 2-39, 5-5, 6-13, 7-1
 - allocating, 7-2, 7-4, 15-4
 - altering, 7-3, 7-5
 - built-in functions, 15-2
 - changing, 2-39
 - default, 6-3
 - freeing, 7-3, 16-6
 - global, 7-4
 - inquiring about, 7-3

- list, 15-1
- predefined, 2-40, 7-1, 7-4
- programmer-defined, 2-40, 7-4
- reserved, 7-4
- table, 2-41, 5-5
- temporary, 7-5
- displaytype field option, 6-6, **6-13**
- DISPLAYTYPE NAME display-type option, 4-37
- displaytypes form option, 2-40, **5-5**
- Dollar sign (\$), 4-46, 9-5

E

- Edit buffer, 4-10
- edit_form command, 1-5, 4-2, **B-2**
- Edit mode, 4-12
 - changing, 4-12, 4-23
 - initial, 4-41, 7-14
- Edit request level, 4-8
- Edit requests, 4-12
- Editors, 4-2
- En/disable line number mode request, 4-23
- En/disable overlay mode request, 4-12, 4-23
- En/disable request menu display request, 4-21
- Enter field request, 4-22, 4-55
- (ENTER) request, 2-21, 2-30, 3-13, 4-16
- Entering a form. *See* Submitting a form
- Error codes, 5-18, 10-1
 - e\$form_aborted (1453), 3-14
 - e\$form_invalid_3270_action (3889), 5-14
 - e\$form_invalid_3270_option (3888), 5-14
 - e\$form_needs_input_field (3918), 3-16
 - e\$invalid_form_id (3794), F-4
 - e\$line_hangup (1365), 10-2
 - e\$parity_error (2916), 10-2
 - e\$timeout (1081), 3-15
- error condition, 10-1
- ERROR MESSAGE FIELD form option, 4-50
- Errors
 - handling, 10-1
 - status form option, 5-18
 - validation, 10-4
- Exact form, 2-20, 4-25

F

- Field definition files, 4-22, 4-23, **4-54**
- Field definitions directory, 4-6, 4-22, 4-23
- Field descriptions, 6-1

- Field entry trap, 3-14, 7-14, **13-2**
- Field exit trap, 3-14, 7-13, **13-3**
- FIELD_HAS_CHANGED data-state switch, 8-7
- Field-ID constants, 2-26, 4-54, 4-58
- Field-ID names, 4-58
- Field IDs, 3-1, 4-54
 - allocating, 6-4, 15-4
 - freeing, 6-7
 - list, 15-1
 - maximum, 5-11
- Field-IDs file, 1-6, 2-9, **4-54**, 4-58
 - prefixes, 2-19, 4-6, 4-44
 - using, 2-26, 4-58, 4-60
- Field modes, 5-12, 6-16, A-5
 - initializing, A-6
 - modifying, A-6
- FIELD NAME field option, 2-10, 4-30
- FIELD NAME window field option, 4-42
- Field options, 4-29, 4-58, 6-8
- Field pictures, 6-18, 7-18, 9-1, 10-4
- FIELD TYPE field option, 4-31
- FIELD_VALUE_GIVEN data-state switch, 2-39, **8-7**
- Field-value variables, 1-2, 2-26
 - array field, 4-62, 6-10
 - declaring, 2-26, 4-53, 4-61
- Field-values file, 1-6, 2-9, **4-53**, 4-61
 - excluding fields, 4-32
 - sequence of entries, 4-33
 - using, 2-26, 4-61
- field-values sequence field option, 4-33
- Fields, 1-1, 3-1
 - allocating, 6-4, 15-4
 - alphanumeric, 3-3
 - altering, 4-24, 4-26, 6-6
 - array, 4-27, 4-30, 6-9
 - built-in functions, 15-3
 - character sets, 3-7, 4-32, 4-39, 6-22, 6-23, 7-15
 - creating, 2-10, 4-22, 4-25, 4-26, 6-2
 - cycle, 1-3, 2-15, 4-37, 7-7
 - data states, 1-3, 2-36, 5-3, 6-11, 8-1
 - data types, 2-11, 4-33
 - deleting, 1-7, 4-21, 6-7, 16-6
 - disabling, 4-27, 4-31, 8-6
 - display types, 1-3, 2-39, 5-5, 6-13, 7-1
 - inquiring about, 6-6
 - justification, 4-38
 - layout, 2-3
 - length, 2-11, 4-31, 6-16

- naming, 2-10, 4-30, 6-18
- null values, 1-2, 3-4
- numeric, 3-2, 9-4
- optional, 2-3
- output values, 3-5
- position, 2-11, 4-30, 4-42, 6-19
- precision, 4-35, 9-3
- predefined, 1-6
- required, 2-3, 2-11, 2-39, 4-31, 8-9, 10-4
- shift modes, 3-7, 4-32, 6-22, 6-23
- simple, 4-26
- uncommitted, 4-26, 4-27
- validating, 4-37, 10-4
- video attributes, 2-22, 4-38, 7-20
- window, 4-22, 4-41, 6-25, 11-1
- FILTER_FOR_CONVERSION** data-state switch, 8-7
- Filtering, 8-7, 9-1, 9-4, 10-4
 - input values, 9-7
 - output values, 9-6
- find_screen_field** function, 15-5
- first_changed_field** function, 15-5
- first_screen_displaytype** function, 15-6
- first_screen_field** function, 15-6
- FMS**. *See* Forms Management System (FMS)
- FORCE_INSERT_MODE** display-type option, 4-41
- FORCE_INSERT_MODE** display-type switch, 7-14
- FORCE_OVERLAY_MODE** display-type option, 4-41
- FORCE_OVERLAY_MODE** display-type switch, 7-14
- Form definition file, 1-6, 4-5, 4-6, 4-53, 4-56
- form form option, 5-5
- Form header, E-1
- Form IDs, 2-28, 5-6
- Form layout, 2-3
- Form name, 2-8, 2-18, 4-6, 4-44, 5-13
- Form object module, 1-3, 1-6, 2-24, 2-42, 4-53
- Form options, 2-18, 4-43, 4-58, 5-1
- Form picture, 4-58
- Form reference counts, 12-2
- Form specifier, 2-28, 5-1
- .form suffix, 4-5, 4-6, 4-53
- formid form option, 2-28, 5-6
- Forms, 1-1
 - activating, 3-10
 - caching, 3-12, 12-1, 16-19
 - canceling, 1-2, 3-14
 - converting to new-style, C-1
 - defining, 2-8, 4-1, 6-2
 - discarding, 3-12, 12-2, 16-8
 - displaying, 2-28, 3-10, 16-12, 16-17
 - editing, 1-5, 4-1
 - initializing, 2-27, 3-10, 16-9
 - inquiring about, 3-11, 16-15
 - long, 2-17, 13-5
 - naming, 2-8, 2-18, 4-44, 5-13
 - old-style, B-1, C-1
 - output-only, 3-16
 - planning, 2-2
 - predefined, 1-6
 - renaming, 4-7
 - sizes, 2-3, E-1
 - submitting, 1-2, 3-13
- Forms Editor, 1-3, 1-5, 2-8, 4-1
 - altering fields, 2-12, 4-22, 4-24, 4-26
 - background text, 4-11
 - batch process, 4-7
 - creating background text, 2-9
 - creating fields, 2-10, 4-22, 4-25, 4-26
 - data states, 8-2
 - direct edit requests, 4-13, D-2
 - display-type options, 4-36
 - field options, 4-29
 - form options, 4-43
 - invoking, 2-8, 4-4
 - menu edit requests, 4-21, D-1
 - new versus old, 4-2
 - old, B-1
 - quitting, 2-25, 4-23
 - retrieving field definitions, 4-23, 4-55
 - saving field definitions, 4-22, 4-55
 - writing the form, 2-24, 4-25
- Forms input mode, 2-42, 13-10, 14-1, F-1
- Forms Management System (FMS), 1-1
- Forms Processor, 1-3, 1-8
- FORTTRAN** field option, 4-33
- FORTTRAN** form option, 4-51
- FORTTRAN** strings, 4-8, 4-51
- Function keys, 2-19, 4-45, 5-9
- functionkey form option, 5-7, 13-2, 13-3

G

- Generic function keys, 2-19, 4-45, 5-9
- Generic input request codes, 5-7
- Generic input requests, D-1
- GET_INFO_OPCODE** (201), F-3
- getcurs form option, 5-7, 13-2, 13-3
- given field option, 6-14, A-10

Given switch, 2-39, 8-7
 Global control operations, F-1
 Global display types, 7-4
 allocating, 15-4
 list, 15-1
 programmer-defined, 7-4
 reserved, 6-3, 7-4
 Global replace request, 4-22
GO TO **↓** request, 4-16
GO TO **←** request, 4-16
GO TO **→** request, 4-16
GO TO **↑** request, 4-16
GO TO **COLUMN** request, 4-16
GO TO **LINE** **↓** request, 4-16
GO TO **LINE** **↑** request, 4-16
GO TO **LINE** request, 4-16
GO TO **LINE** **(RETURN)** request, 4-17
GO TO **(MARK)** request, 4-17
 Grouping character, 9-3, 9-4, 9-6

H

Hangul character set, 3-6
 Heap, 3-9
 HELP field option, 2-11, 4-33
 help field option, 6-6, 6-14
 Help messages, 2-11, 2-21, 4-33, 6-14
(HELP) request, 2-21, 4-17, 4-33
 HIGH_INTENSITY mode switch, A-5
 HIGH_INTENSITY_VISUAL display-type switch, 7-21
 Highlighting, 4-26, 4-42, 4-51
 canceling, 4-14
 starting, 4-17
 Hyphen (-), 3-3, 9-2, 9-3, 9-6

I

ICSS. *See* International Character Set Support (ICSS)
 icss_edit_form command, 1-5, 2-8, 4-2, 4-4
 icss_fms_object_library directory, 2-42, C-11
 if statements, 2-30
 Immediate return. *See* Traps, on field exit
 IMMEDIATE_RETURN mode switch, A-5
 IN FIELD-VALUES field option, 4-32
 Incidental data, 2-5
 .incl.cobol suffix, 4-53, 4-54

Include files, 1-6, 2-24
 character sets, 3-7
 field-IDs file, 1-6, 2-9, 4-54, 4-58
 field-values file, 1-6, 2-9, 2-26, 4-53, 4-61
 form data state, 2-36, 8-3
 form display type, 2-40
 form options, 2-38
 Include library, 2-42
 INDEX form option, 4-50
 INDEXED_CYCLE_LIST display-type option, 4-40
 INDEXED_CYCLE_LIST display-type switch, 7-9, 7-15
 INITIAL_DISPLAY form option, 4-46
 Initial display versus redisplay, A-2
 INITIAL field option, 4-33
 initial field option, 6-6, 6-15
 Initial output values, 3-5
 accept statement, A-10
 altering, 2-28
 INPUT_DISABLED mode switch, A-5
 INPUT_FIELD data-state switch, 3-3, 8-8
 input field option, 6-6, 6-15
 Input fields, 1-2, 2-3, 3-3, 4-31, 8-8
 Input information, 2-5
 Input time limit, 3-15, 4-48, 5-18
(INSERT DEFAULT) request, 4-17
 Insert literal request, 4-11, 4-23
 Insert mode, 4-12, 4-41, 7-14
(INSERT/OVERLAY) request, 4-12
(INSERT SAVED) **(DISCARD)** request, 4-17
(INSERT SAVED) request, 4-17
 Insert window field request, 4-22, 4-41
 form, 4-42
 Intensity
 background text, 2-23, 4-24, 4-50, 4-51
 fields, 2-22, 4-38
 required fields, 2-22, 4-50
 INTENSITY display-type option, 2-14, 4-38
 International Character Set Support (ICSS), 3-6, 4-32, 4-39, 6-22, 6-23, 7-15
 into form option, 2-28, 5-8
 versus update form option, A-4
 INVERSE mode switch, A-5
 INVERSE_VISUAL display-type switch, 7-21
 Inversion
 background text, 2-23, 4-24, 4-50, 4-51
 fields, 2-22, 4-38
 required fields, 2-22, 4-50

J

JUSTIFICATION display-type option, 4-38
 Justification display-type switches, 7-22
 Justification field options, 6-10, 6-15, 6-21, A-9

K

Kanji character set, 3-6
 Katakana character set, 3-6

Keys

ALPHA LOCK, 4-12
BACK SPACE, 4-13
BACK TAB, 4-13
CANCEL, 2-21, 3-14, 4-14
ENTER, 2-21, 2-30, 3-13, 4-16
 function, 2-19, 4-45, 5-9
HELP, 2-21, 4-17, 4-33
RETURN, 4-18
SHIFT, 4-12
SPACE BAR, 4-12

Keystrokes file, 4-54

keyused form option, 2-29, 5-8

KNOCK_DOWN_FORM_OK_OPCODE (264), 3-16, F-4

KNOCK_DOWN_FORM_OPCODE (240), 3-16, F-4

L

L picture character, 3-3, 9-2
 Labels. *See* Background text
 last_screen_displaytype function, 15-7
 last_screen_field function, 15-7
 Latin alphabet No. 1 character set, 3-6
 Layout of form, 2-3
 left field option, 6-15, A-9
 LEFT_JUSTIFY_FIELD_DATA display-type switch, 7-22
 LENGTH field option, 2-11, 4-31
 length field option, 6-6, 6-16
 Library of field definitions, 4-6, 4-22, 4-23
LINE FEED request, 4-17
 Line numbers, 4-23
 Literal characters, 4-23
 Long forms, 2-17, 13-5
 Loops, 2-31
 LOW_INTENSITY mode switch, A-5
 LOW_INTENSITY_VISUAL display-type switch, 7-21

M

MARK request, 2-23, 4-17
 MASKKEYS form option, 2-19, 4-45
 maskkeys form option, 5-9
 Master forms, 11-1
 Master window, 11-1
 max_displaytype_id form option, 5-11
 max_field_id form option, 5-11
 Menu
 alternative to requests, D-2
 disabling, 4-21
MENU A request, 4-21
MENU D request, 4-21
MENU E request, 4-22, 4-55
 Menu edit requests, 1-6, 4-21, D-1
MENU F request, 2-10, 4-22, 4-25
 form, 2-10, 4-28
 old-style, B-7
MENU G request, 4-22
MENU I request, 4-22, 4-41
 form, 4-42
MENU L request, 4-11, 4-23
MENU N request, 4-23
MENU O request, 4-12, 4-23
MENU Q request, 2-25, 4-23
MENU R request, 4-23, 4-55
MENU request, 4-17, 4-21
MENU S request, 2-18, 4-24, 4-43
 form, 2-18, 4-43
 old-style, B-10
MENU U request, 4-24
MENU V request, 2-23, 4-24, 4-51
 form, 2-23, 4-51
MENU W request, 2-24, 4-25
MENU X request, 2-17, 2-20, 4-25
MENU Z request, 4-25
 MESSAGE form option, 4-50
 message form option, 5-11
 Minus sign (-), 9-5, 9-6
 mode field option, 6-16, A-6
 Modes. *See* Field modes
 modes form option, 5-12, A-6
 Modes table, A-6

N

name field option, 6-6, 6-18
 name form option, 5-13

NEW_DATA_IN_FIELD data-state switch, 8-8
 New Forms Editor, 4-2, 4-4
 next_changed_field function, 15-8
 next_screen_displaytype function, 15-9
 next_screen_field function, 15-10
 nextcursor form option, 5-13, 13-2, 13-3
 NO_COPY_UPDATE options switch, 5-14, 5-19, 6-24
 NO_OVERLAY mode switch, A-5
 Noneditable fields. *See* Defaults, disappearing
 NOT_EDITABLE mode switch, A-5
 NOTRIM_FIELD_DATA_SPACES display-type switch,
 7-22
 Null field values, 1-2, 3-4
 Numeric fields, 3-2, 9-4
 Numeric pictures, 9-3

O

.obj suffix, 2-24, 4-53
 Object library, 2-42, C-11
 Object module. *See* Form object module
 Obsolete features, A-1
 Old Forms Editor, 4-2
 Optional fields, 2-3, 4-31
 cycle, 2-16, 7-9
 options form option, 2-38, 5-13
 initial value, 2-38
 origin form option, 5-16, 11-2
 Output fields, 1-2, 2-2, 3-3, 4-31
 Output information, 2-2
 Output-only fields, 2-14, 3-4, 4-31, 6-2
 Output-only forms, 3-16
 Overlay mode, 4-12, 4-23, 4-41, 7-14

P

PASCAL field option, 4-33
 PASCAL form option, 4-51
 perform screen delete statement, 1-7, 3-11, 6-7,
 7-3, 16-6
 perform screen discard statement, 1-7, 3-12,
 12-2, 16-8
 perform screen initialization statement, 1-7,
 2-27, 3-10, 16-9
 without predefined form, 6-2
 perform screen input statement, 1-7, 1-8, 2-28,
 3-10, 16-12
 perform screen inquire statement, 1-7, 2-38,
 3-11, 6-6, 7-7, 7-11, 16-15

perform screen output statement, 1-7, 3-10,
 16-17
 perform screen save statement, 1-7, 3-12, 12-1,
 16-19
 perform screen update statement, 1-7, 3-11, 6-6,
 16-21
 Period (.), 3-2, 9-2, 9-3
 Picture characters, 3-2, 3-3, 6-19, 7-18, 9-2
 PICTURE display-type option, 2-11, 4-37, 9-3
 picture display-type option, 7-18, 9-1, 9-3
 picture field option, 6-18, A-10
 Pictures, 7-18, 9-1, 10-4
 alphanumeric, 3-3, 9-3
 numeric, 3-2, 9-3
 PL/1 field option, 2-11, 4-33
 PL/1 form option, 2-19, 4-51
 Planning the form, 2-2
 Plus sign (+), 9-5
 .pm suffix, 2-42
 portid form option, 5-16
 Ports, 5-16
 POSITION field option, 2-11, 4-30
 position field option, 6-6, 6-19, 11-2
 POSITION window field option, 4-42
 Precision of numeric fields, 4-35, 9-3
 Predefined display types, 2-40, 7-1, 7-4
 Predefined fields, 1-6
 Predefined forms, 1-6
 referencing, 2-28, 5-5
 Prefix for form, 2-19, 4-6
 PREFIX form option, 2-19, 4-44
 prev_screen_displaytype function, 15-11
 prev_screen_field function, 15-12
 PRODUCE INTO form option, 2-19, 4-46
 putcursor form option, 2-32, 5-17

Q









Quit request, 2-25, 4-23





R

range display-type option, 7-19
 range field option, 6-20, A-10
 Range restrictions, 4-37, 7-19, 10-4
 Read field request, 4-23, 4-55
 Recompiling, 2-24
 redisplay form option, 5-17, A-3, C-4
 (REDISPLAY) (REDISPLAY) request, 4-18

REDISPLAY request, 4-17
 redisplayfield field option, 6-20, A-10, C-4
 Reference counts, 12-2
 Renaming a form, 4-7
 replace compile-time statement. *See* %replace
 compile-time statement
 REQ FIELD MODE TOGGLES form option, 4-50
 REQUIRED_FIELD data-state switch, 2-39, 8-9, 10-4
 REQUIRED field option, 2-11, 4-31
 required field option, 6-21, A-10
 Required fields, 2-3, 2-11, 2-39, 4-31, 8-9, 10-4
 video attributes, 2-22, 4-50
 Reserved display types. *See* Global display types
RETURN request, 4-18
 right field option, 6-21, A-9
 Right graphic set, 3-6, 4-32
 RIGHT_JUSTIFY_FIELD_DATA display-type switch,
 7-22
 ROW SPACING field option, 4-30

S

s\$begin_forms_input subroutine, 14-3
 s\$control subroutine, 3-16, F-1
 s\$end_forms_input subroutine, 14-4
SAVE request, 4-18
 screen_field_position function, 15-13
 screen statements, 1-3, 1-6, 16-1
 display-type descriptions, 7-2
 field descriptions, 6-1
 form options, 5-1
SCROLL  request, 4-18
SCROLL  request, 4-18
SCROLL  request, 4-18
SCROLL  request, 4-18
SCROLL **SHIFT**  request, 4-18
SCROLL **SHIFT**  request, 4-18
SCROLL **SHIFT**  request, 4-19
SCROLL **SHIFT**  request, 4-19
 Scrolling
 between forms, 13-5
 on a screen, 4-46, 5-16
 within a form, 13-6
 Search paths
 include library, 2-42
 object library, 2-42, C-11
 Self-insertion characters, 9-3
 Set bell column request, 4-25
 SET_INFO_OPCODE (202), F-1

SET_MODES_OPCODE (207), F-1
 Set/modify form options request, 2-18, 4-24,
 4-43
 form, 2-18, 4-43
 old-style, B-10
SHIFT  request, 4-19
SHIFT  request, 4-19
SHIFT  request, 4-19
SHIFT  request, 4-19
 SHIFT, DCS = field option, 4-32
 shift field option, 3-8, 6-22
SHIFT key, 4-12
SHIFT **STATUS** request, 4-19
 Show exact form request, 2-17, 2-20, 4-25
 Sign characters, 9-5, 9-6
 Signaled conditions, 10-1
 Simple fields, 4-26
 Single-shift characters, 3-7
 Sizes of forms, 2-3, E-1
 Slant (/), 3-3, 9-2, 9-3, 9-5, 9-6
SPACE BAR key, 4-12
 Space characters, 2-4, 9-5, 9-6
 Special numeric characters, 9-5
 Special switches
 action, 7-15
 option, 5-15
 visual, 7-22
 Statements, 1-6, 16-1
 accept, 16-2, A-1
 perform screen delete, 1-7, 3-11, 6-7, 7-3,
 16-6
 perform screen discard, 1-7, 3-12, 12-2, 16-8
 perform screen initialization, 1-7, 2-27,
 3-10, 6-2, 16-9
 perform screen input, 1-7, 1-8, 2-28, 3-10,
 16-12
 perform screen inquire, 1-7, 2-38, 3-11,
 6-6, 7-7, 7-11, 16-15
 perform screen output, 1-7, 3-10, 16-17
 perform screen save, 1-7, 3-12, 12-1, 16-19
 perform screen update, 1-7, 3-11, 6-6, 16-21
 status form option, 2-28, 2-29, 5-18
STATUS request, 4-19
 Storage sizes, E-1
 STRINGS form option, 4-51
 SUB-FORM SIZE window field option, 4-43

Subforms, 11-1
 defining, 11-1
 displaying, 11-3
 initializing, 11-2
 sizes, 4-43, 11-2
 Submitting a form, 1-2, 3-13
 Subroutines, 14-1
 s\$begin_forms_input, 14-3
 s\$control, F-1
 s\$end_forms_input, 14-4
 Supplemental character sets, 3-6, 4-11, 4-32,
 6-22, 6-23, 7-15
 Symbol table, 4-8

T

(TAB) request, 4-19
 Temporary display types, 7-5
 terminal port, 5-16
 Terminal types, D-1
 attribute requests, D-5
 input requests, D-1
 output requests, D-4
 Terminals
 3270, 4-49, 5-14
 function keys, 2-19, 4-45, 5-9
 requirements, D-1
 type table, D-2
 V101, 2-3
 V102, 2-3
 Testing a form, 2-17, 2-20, 4-25
 TIME OUT form option, 4-48
 timeout form option, 3-15, **5-18**
 Toggling video attributes
 background text, 2-23, 4-50
 required fields, 2-22, 4-50
 Trap field, 13-1
 TRAP mode switch, A-5
 TRAP ON FIELD ENTRY display-type option, 4-40
 TRAP_ON_FIELD_ENTRY display-type switch, 3-14,
 7-14, 13-2
 TRAP ON FIELD EXIT display-type option, 4-40
 TRAP_ON_FIELD_EXIT display-type switch, 3-14,
 7-13, 13-3
 Traps, 2-6, 3-14, 4-40, 4-49, 7-13, 7-14, **13-1**
 on field entry, 3-14, 4-40, 7-14, **13-2**
 on field exit, 3-14, 4-40, 7-13, **13-3**
 vertical scroll, 3-15, 4-49, 5-14, **13-5**
 TRIM BLANKS display-type option, 4-38

Type ahead. *See* Forms input mode

U






U picture character, 3-3, 9-2
 Uncommitted fields, 4-26, 4-27
 UNDERLINED mode switch, A-5
 UNDERLINED_VISUAL display-type switch, 7-21
 Underlining
 background text, 2-23, 4-24, 4-50, 4-51
 fields, 2-22, 4-38
 required fields, 2-22, 4-50
 unshift field option, 3-8, **6-23**
 UNSHIFT, RGS = field option, 4-32
 update field option, 6-2, 6-4, 6-6, **6-24**
 Update fields request, 4-24
 update form option, 2-29, **5-19**, 6-4
 versus into form option, A-4
 Update switches, 7-5
 action, 7-12
 visual, 7-21
 User actions, 1-2, 2-5, 3-13
 User heap, 3-9

V

VALIDATE display-type option, 4-37
 validate display-type option, **7-19**, 10-5
 VALIDATE_ERRORS_OFF options switch, **5-14**, 13-3
 validate field option, **6-24**, A-10
 VALIDATE form option, 4-48
 VALIDATE_ONE_FIELD options switch, **5-15**, 13-3
 Validating field values, 10-4, 13-3
 Validation routines, 7-19, 10-4, **10-5**
 Validation suite, 10-4
 VALUE RESTRICTION display-type option, 2-15, 4-37
 Variables. *See* Field-value variables
 Vertical scroll trap, 3-15, 4-49, 5-14, **13-5**
 VERTICAL_SCROLL_TRAP form option, 4-49
 VERTICAL_SCROLL_TRAP options switch, 3-15,
 5-14, 13-5
 Video attributes, 2-21
 of background text, 2-23, 4-24, 4-50, 4-51
 of fields, 2-22, 3-4, 4-38, 7-20
 of required fields, 2-22, 4-50
 Visual attributes, 2-40, 7-20
 blanking, 7-21
 blinking, 7-21
 intensity, 7-21

inversion, 7-21
 justification, 7-22
 underlined, 7-21
 visual display-type option, 7-5, 7-20

W

WIDE CURSOR form option, 4-49
 WIDE_CURSOR options switch, 5-14
 window field option, 6-6, 6-25, 11-2
 Windows, 2-3, 11-1
 defining, 4-22, 4-41, 6-25, 11-1
 master, 11-1
 referencing, 5-16, 11-2
 sizing, 11-2
 WORD  request, 4-20
 WORD  request, 4-20
 WORD CHANGE CASE  request, 4-20
 WORD CHANGE CASE  request, 4-20
 WORD CHANGE CASE  request, 4-20
 Write request, 2-24, 4-25

X

X picture character, 3-3, 9-2

Z

Z picture character, 3-2, 9-2, 9-3
 Zero suppression, 3-2, 9-2

)

)

)

)

)

