

# VSI OpenVMS

## Introduction to VSI FMS

**Operating System and Version:** VSI OpenVMS IA-64 Version 8.4-1H1 or higher  
VSI OpenVMS Alpha Version 8.4-2L1 or higher  
VSI OpenVMS x86-64 Version 9.2-1 or higher

**Software Version:** VSI FMS Version 2.6 or higher

---

## Introduction to VSI FMS



VMS Software

---

Copyright © 2025 VMS Software, Inc. (VSI), Boston, Massachusetts, USA

### Legal Notice

Confidential computer software. Valid license from VSI required for possession, use or copying. Consistent with FAR 12.211 and 12.212, Commercial Computer Software, Computer Software Documentation, and Technical Data for Commercial Items are licensed to the U.S. Government under vendor's standard commercial license.

The information contained herein is subject to change without notice. The only warranties for VSI products and services are set forth in the express warranty statements accompanying such products and services. Nothing herein should be construed as constituting an additional warranty. VSI shall not be liable for technical or editorial errors or omissions contained herein.

HPE, HPE Integrity, HPE Alpha, and HPE Proliant are trademarks or registered trademarks of Hewlett Packard Enterprise.

DEC, DEC/CMS, DEC/MMS, DECnet, DECsystem-10, DECSYSTEM-20, DECUS, DECwriter, MASSBUS, MICRO/PDP-11, Micro/RSX, MicroVMS, PDP, PDT, RSTS, RSX, TOPS-20, UNIBUS, VAX, VMS, VT, and *mm* are trademarks or registered trademarks of Hewlett Packard Enterprise.

# Table of Contents

<b>Preface .....</b>	<b>vii</b>
1. About VSI .....	vii
2. Document Structure .....	vii
3. Intended Audience .....	viii
4. VSI Encourages Your Comments .....	viii
5. OpenVMS Documentation .....	viii
6. Conventions .....	viii
<b>Chapter 1. Overview of FMS .....</b>	<b>1</b>
1.1. FMS Components .....	2
1.1.1. Form Editor .....	2
1.1.2. Form Language Translator .....	2
1.1.3. Form Librarian .....	2
1.1.4. Form Driver .....	2
1.1.5. Sample Application .....	3
1.1.6. Form Tester .....	3
1.1.7. Form Application Aids .....	3
1.2. FMS Application Development .....	3
1.3. Reading Path for the FMS Document Set .....	9
<b>Chapter 2. Running the Sample Application Program .....</b>	<b>13</b>
2.1. Setting Up Your Terminal .....	13
2.2. Starting the Sample Application .....	13
2.3. Printing SAMPCH.DAT .....	19
<b>Chapter 3. Creating Forms .....</b>	<b>21</b>
3.1. Setting Up Your Terminal .....	21
3.2. Examining Two Forms from the Sample Application .....	21
3.2.1. MENU Form .....	22
3.2.2. DEPOSIT Form .....	23
3.3. Creating MENU and DEPOSIT .....	24
3.3.1. Creating MENU .....	25
3.3.1.1. Assigning Form Attributes: The Form Phase .....	26
3.3.1.2. Laying Out the Form: The Layout Phase .....	27
3.3.1.3. Assigning Field Attributes: The Assign Phase .....	32
3.3.1.4. Testing a Form: The Test Phase .....	34
3.3.1.5. Saving a Form: The Exit Phase .....	34
3.3.2. Creating DEPOSIT .....	34
3.3.2.1. Assigning Form Attributes: The Form Phase .....	35
3.3.2.2. Laying Out the Form: The Layout Phase .....	35
3.3.2.3. Assigning Field Attributes to DEPOSIT: The Assign Phase .....	37
3.3.2.4. Alternative Method of Assigning Field Attributes .....	39
3.3.2.5. Testing a Form: The Test Phase .....	39
3.3.2.6. Saving a Form: The Exit Phase .....	39
3.4. Creating a Help Form .....	39
3.4.1. Creating HELP_MENU .....	40
3.4.2. Associating HELP_MENU with MENU .....	41
<b>Chapter 4. Creating a Form Library .....</b>	<b>43</b>
4.1. Create a Library File .....	43
4.2. Obtain a Library Directory Listing .....	44
4.3. Interpret a Form Description .....	45

4.4. Obtain a Form Image .....	46
<b>Chapter 5. Writing an Application .....</b>	<b>49</b>
5.1. Form Driver Concepts .....	49
5.1.1. Form Driver Calls .....	50
5.1.1.1. Functional Division of Calls .....	50
5.1.1.2. Form Driver Calls Used in the Subset Application .....	51
5.1.2. String Handling .....	51
5.2. Sample Application Subset .....	52
5.2.1. The Main Program .....	52
5.2.2. Subset Subroutines .....	53
5.2.3. Preparing the Main Program .....	53
5.2.3.1. Initializing Calls .....	54
5.2.3.2. Coding the Body of the Main Module .....	56
5.2.3.3. Closing Calls .....	56
5.2.4. Coding the INACCT Subroutine .....	57
5.2.5. Coding the MENU Subroutine .....	58
5.2.6. Coding the WRITCH Subroutine .....	61
5.2.7. Coding the MAKDEP Subroutine .....	61
5.2.8. Coding Subroutines VUERE and VUEACT .....	63
5.2.9. Coding Subroutine GETSTA .....	64
<b>Chapter 6. Compiling, Linking, and Running an Application .....</b>	<b>67</b>
6.1. Compiling the Subset .....	67
6.2. Linking the Subset .....	67
6.3. Running the Subset .....	68
<b>Chapter 7. Programming Features .....</b>	<b>69</b>
7.1. Indexing .....	69
7.1.1. Create REGISTER .....	71
7.1.2. Assign Indexing Attributes .....	71
7.1.3. Manipulate Indexed Fields in the VUERE Subroutine .....	72
7.2. Scrolling .....	73
7.2.1. Complete the Check Register Form, REGISTER .....	73
7.2.2. Insert REGISTER in SUBSET.FLB .....	75
7.2.3. Writing the Statements that Support Scrolling .....	75
7.3. Named Data .....	84
7.4. User Action Routines .....	87
7.4.1. Field Completion UARs .....	88
7.4.2. Help UARs .....	88
7.4.3. Function Key UARs .....	88
7.4.4. Creating a Sample UAR .....	88
<b>Chapter 8. Advice to New Users .....</b>	<b>95</b>
8.1. Good Form Design .....	95
8.1.1. Sorting Information .....	95
8.1.2. Providing a Title .....	96
8.1.3. Writing Good Captions .....	96
8.1.4. Using Check Boxes .....	96
8.1.5. Using Reverse Video Screen Characteristics .....	97
8.1.6. Providing Instructions .....	97
8.2. Use of the Video Screen .....	98
8.2.1. How to Present Data .....	98
8.2.2. Screen Layout .....	100

8.2.3. Communication with the Operator ..... 100

8.2.4. Recovery Procedures ..... 101

8.3. FMS Field Attributes ..... 101

**Appendix A. Subset Application Listing ..... 103**



# Preface

The *Introduction to VSI FMS* gives an overview of the Forms Management System and provides exercises for using many of the features of FMS. After reading this manual, you should be able to use the rest of the FMS documentation and the FMS software easily.

This manual does the following:

- Introduces FMS and describes its components
- Walks you through the Sample Application, a program provided in the FMS kit to demonstrate most of the FMS features
- Shows you how to create a subset of the Sample Application
- Describes some of the more advanced features of FMS and shows you how to use them
- Gives hints on designing good forms

## 1. About VSI

VMS Software, Inc. (VSI) is an independent software company licensed by Hewlett Packard Enterprise to develop and support the OpenVMS operating system.

## 2. Document Structure

*Chapter 1, "Overview of FMS"* introduces FMS and gives a general description of its components. The chapter tells how to use this manual and also provides a reading path for the FMS document set.

*Chapter 2, "Running the Sample Application Program"* gives you an opportunity to see an FMS application. Specific capabilities of FMS are pointed out as you run the Sample Application.

*Chapter 3, "Creating Forms"* is an exercise in designing, creating, and testing two forms used in the Sample Application.

*Chapter 4, "Creating a Form Library"* describes how to create a library for the forms that you created in *Chapter 3, "Creating Forms"* and how to obtain and use form descriptions of those forms.

*Chapter 5, "Writing an Application"* is an exercise in coding a subset of the Sample Application. The exercise uses the two forms you created in *Chapter 3, "Creating Forms"*. In *Chapter 5, "Writing an Application"*, you use the Form Driver.

*Chapter 6, "Compiling, Linking, and Running an Application"* describes how to compile, link, and run the subset application created in *Chapter 5, "Writing an Application"*. The steps outlined here are described in general terms so that you can apply them to your FMS applications.

*Chapter 7, "Programming Features"* describes some of the more advanced features of FMS, such as scrolling, Named Data, indexing, and user action routines.

*Chapter 8, "Advice to New Users"* describes a successful form and explains how to incorporate good form design into your forms.

## 3. Intended Audience

This manual is directed to two groups of readers:

- Those who want a general overview of FMS Version 2
- Those who have had experience with FMS Version 1 software and wish to learn about FMS Version 2

Readers should be familiar with BASIC and DCL.

## 4. VSI Encourages Your Comments

You may send comments or suggestions regarding this manual or any VSI document by sending electronic mail to the following Internet address: <docinfo@vmsssoftware.com>. Users who have VSI OpenVMS support contracts through VSI can contact <support@vmsssoftware.com> for help with this product.

## 5. OpenVMS Documentation

The full VSI OpenVMS documentation set can be found on the VMS Software Documentation webpage at <https://docs.vmsssoftware.com>.

## 6. Conventions

The following conventions are used in this manual:

Convention	Meaning
Uppercase letters	In commands and examples, indicate that the user types the item exactly as shown.
Brackets []	Indicate that the item is optional.
Red print	Indicates what the user types.
CTRUx	Indicates that you simultaneously press the key labeled CTRL and another key.
GOLDx	Indicates that you press the PF1 key before you press the second key.

Unless specified otherwise, you terminate commands by pressing the RETURN key.



# Chapter 1. Overview of FMS

We are all familiar with forms in our everyday lives – tax forms, employment forms, insurance forms, and so on. Most of these forms contain the same two types of information. First, the forms have information already printed on them – titles, headings, descriptions of items to be filled in, and instructions. Second, the forms have blank spaces for the information to be filled in by the user of the form.

Video forms created with OpenVMS FMS Forms Management System look very similar to the paper forms described above. Like paper forms, video forms contain information that never changes; this background text is protected against modification by both the program and the terminal user. The areas of the form that can be changed – by the application program, by the terminal user, or by both – are known as fields. On the surface, then, video forms are like paper forms.

When we look further, however, we find that OpenVMS FMS adds many properties to video forms that have no counterpart in paper forms. The interactive nature of FMS allows data entered by the operator to be validated as it is entered to ensure that letters are not typed where numbers are required, that required fields are not left blank, and so on. Furthermore, FMS provides techniques for additional validation and processing of entered data to be done by the application program. Therefore, the terminal user will see a single smooth interaction with the computer system, with all errors being caught at the earliest possible moment. Other features that FMS presents to the terminal user include the scrolling of portions of the form to show more lines of information than can fit on the screen at once, the display of help information whenever the operator presses the HELP key, and a variety of visual effects ranging from double-high, double-wide characters to forms overlaid on other forms.

## Forms as Seen by the Programmer

To the application developer, forms contain much more than background text and fields. Some attributes, such as the name of a help form, pertain to the entire form; other attributes, such as right justification, pertain to individual fields. Another part of the form definition allows the application developer to specify that the cursor should move from field to field in a sequence other than the normal left-to-right, top-to-bottom sequence. Finally, the programmer can specify the names of subroutines either to be automatically invoked by FMS upon completion of fields or when function keys are depressed or to supplement the standard FMS handling of help requests. In addition to the parameters that can be stored with the form and passed to these subroutines, FMS provides additional parameter storage for other parameters that can be read by the program from the form description during program execution.

FMS provides support for the following languages:

- VAX-11 BASIC
- VAX-11 BLISS
- VAX-11 C
- VAX-11 COBOL
- VAX-11 FORTRAN
- VAX-11 PASCAL
- VAX-11 PL/I

## 1.1. FMS Components

To create forms and to write and run a program, you use the following components:

- Form Editor
- Form Language Translator
- Form Librarian
- Form Driver
- Form Tester
- Form Application Aids

### 1.1.1. Form Editor

The Form Editor lets you design, modify, test, and store forms. When you use the Form Editor, you design and modify forms interactively; your screen always shows the current state of the form you are working on. You can use special keypad and keyboard functions to specify video display characteristics such as boldface type or reverse video for different parts of the form. To help operators, you can include on-line help – short explanations about parts of the form and about the whole form.

When designing forms, you assign names to the forms and to data that will be entered or displayed when the application runs. The *VSI FMS Utilities Reference Manual* describes the Form Editor in detail.

### 1.1.2. Form Language Translator

The Form Language Translator converts form descriptions that you have created with a text editor, using Form Language statements, into binary forms. The Form Language has all the capabilities of the Form Editor, but lets you create forms on any terminal, video or hard copy. If you prefer a language-like method of preparing forms, you can use the Form Language. The Form Translator can also convert form descriptions produced by the Form Application Aids into binary forms. The *VSI FMS Utilities Reference Manual* describes the Form Language and Form Language Translator in detail.

### 1.1.3. Form Librarian

The Form Librarian allows you to create library files in which you can insert, extract, or delete forms. For more information on the Form Librarian, see the *VSI FMS Utilities Reference Manual*

### 1.1.4. Form Driver

The Form Driver is a set of subroutines that your program uses to access the forms that you created with the Form Editor or with the Form Language. FMS applications access forms by means of Form Driver calls that you include in the source program. All Form Driver calls refer to specific forms and data within forms by names that you assign during form editing. When the FMS application runs, the Form Driver does the following:

- Attaches the operator's terminal and establishes I/O channels to the appropriate form library and the terminal

- Displays forms and accepts operator input
- Refers to data specifications contained in the form to check that the operator input is valid
- Responds to operator requests for help by displaying help text associated with the form being processed
- Displays or erases data in the forms

The *VSI FMS Form Driver Reference Manual* describes the Form Driver and its calls in detail.

### 1.1.5. Sample Application

The Sample Application (SAMP) is a demonstration program in the FMS distribution kit. The Sample Application shows most of the features FMS provides and is a learning tool. All the examples and exercises in this manual are taken from the Sample Application. You will run the Sample Application in *Chapter 2, "Running the Sample Application Program"*. The forms you create in *Chapter 3, "Creating Forms"*, Creating Forms, are from the Sample Application. In *Chapter 5, "Writing an Application"*, you will write a subset of the Sample Application, using the forms that you created earlier.

### 1.1.6. Form Tester

The Form Tester lets you test a form without first having to put the form in a library or write an application program. The Form Tester uses the Form Driver to display your form and to perform I/O operations. When using the Form Tester, you see the form as it will appear when the application runs. You can enter data into fields and check the validation of the data entered.

### 1.1.7. Form Application Aids

The Form Application Aids allows you to do the following:

- Convert binary forms to object modules for use as memory-resident forms
- Provide form descriptions
- Create vector modules for user action routines
- Create COBOL data definition files and DATATRIEVE domain definition files

Using the Form Application Aids, you can produce four types of form descriptions:

- Form Language statements that are suitable for translation into binary forms
- Form images either with or without escape sequences
- Field data structure descriptions that are compatible with COBOL and DATATRIEVE applications that use FMS
- Brief summaries of form data

## 1.2. FMS Application Development

The development of an FMS application involves three processes:

- Creating forms and form libraries
- Writing the application's source program
- Writing the application's user action routine(s)

This section discusses each process, showing how and where each component of FMS is involved with the development of the entire application.

Before beginning work on these processes, you must plan the application. This step, often the most important, deserves a great deal of attention. You must first analyze the task that the FMS application is to perform and determine what types of data the application will be working with. When you plan your FMS application, it helps to know what kinds of data the operator is expected to enter at the terminal. Next, plan your forms based on that data. You can then prepare the source code to handle the forms that you will need for your application. Keep in mind the operator's skills and the computer system on which the application will be running. You can provide online help for the operator as he or she runs the application.

The sections that follow provide details on each step of the FMS application development cycle.

### **Create Forms and Form Libraries**

Using the Form Editor, lay out the forms. Because the Form Editor is interactive, you can plan your forms and modify them as you create them. Sketching them beforehand is unnecessary. Arrange the text and video features of the forms so that the forms are orderly and pleasing to the eye. When you are done laying out the forms, you can test the forms to verify that only the desired types of data will be accepted.

If you are creating forms with the Form Language, use a text editor to enter the statements that make up the form, called a source form description. Use the Form Translator to convert the source form description into a binary form description, which is suitable for inserting into a Form Library or for linking with the application's object module as a memory-resident form. You can use the Form Tester to test this binary form description. The Form Tester displays the form as if it were being displayed by the application on the screen during run time. You can try entering data into the form to verify that it accepts and displays data as intended. Source form descriptions, binary form descriptions, and memory-resident forms are discussed in detail later in this manual.

### **Store Forms in a Form Library**

After you have created and tested your forms, use the Form Librarian to store them in a library file. During run time, the application program accesses forms from the form library unless you have made the forms memory resident.

### **Create a User Action Routine Vector Module**

If your application has user action routines (UARs), use the Form Application Aids to create a UAR vector module. UAR vector modules are discussed in *Chapter 7, "Programming Features"*.

### **Create Memory-Resident Forms**

If you want to use memory-resident forms in your application, use the Form Application Aids to convert the forms in your form library into memoryresident format. Note that you do not need to create memory-resident forms directly from the form library. That is, you can create a memory-resident form immediately after it has been created by the Form Editor or the Form Translator. However, all memory-

resident forms used in an application can be linked only as a single file. So if you want to make several forms memory resident, create them directly from the form library.

### **Obtain a Directory of the Form Library**

You can use the Form Application Aids to list the contents of a form library file. This listing shows all the forms in a form library.

### **Obtain a Form Description**

You can use the Form Application Aids to get a form description in source code. Form descriptions are particularly useful as hard-copy reference for forms. Form descriptions are discussed in *Chapter 4, "Creating a Form Library"*. Note that the source form description can be input to the Form Translator to create a binary form, suitable for insertion into a form library or for use as a memory resident form.

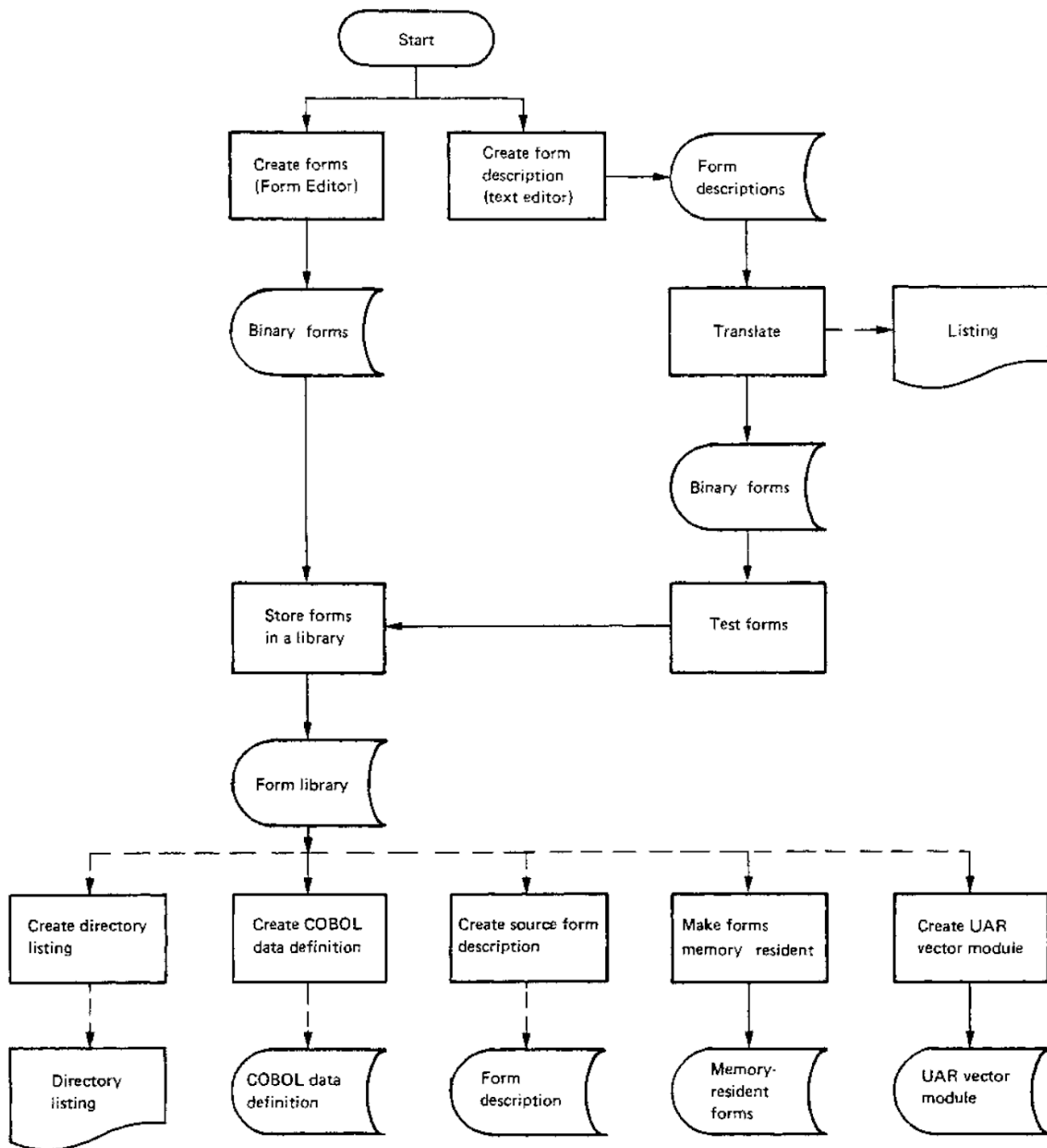
### **Obtain COBOL Data Definition and DATATRIEVE Domain Definition Files**

You can use the Form Application Aids to create a COBOL data declaration file for the forms in your form library if your application is written in VAX-11 COBOL. The COBOL data declaration file can be compiled with the source file for your application. Some editing to this file may be necessary.

*Figure 1.1, "Form Development Cycle"* is a flowchart that shows the development of forms and libraries.

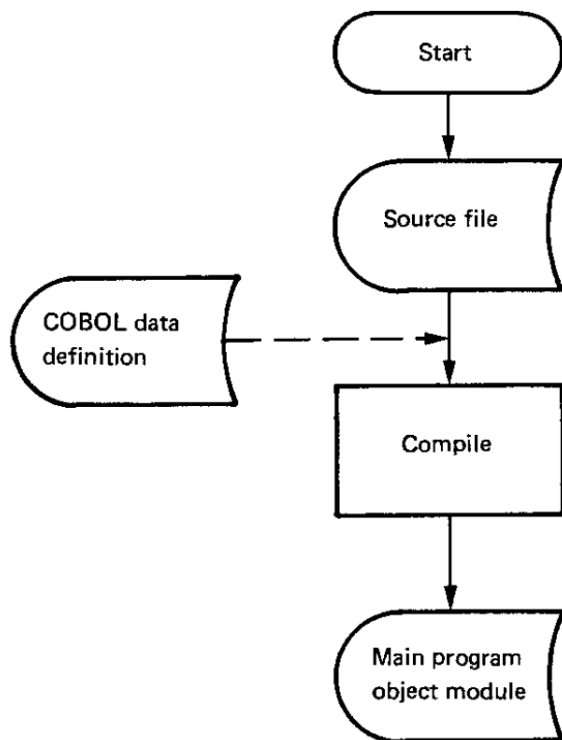
### **Write the Application's Source Program**

Referring to the form descriptions produced by the Form Application Aids, write the source code for your application, using a text editor. Refer to the form descriptions for the correct names of the forms and fields in the form. Include Form Driver calls in your source program to display forms and to perform I/O transfers among the terminal, the form library, and the application.

**Figure 1.1. Form Development Cycle****Compile the Application**

Compile the application's source program to produce an object module. If your application is written in VAX-11 COBOL, compile the COBOL data definition file with your source program.

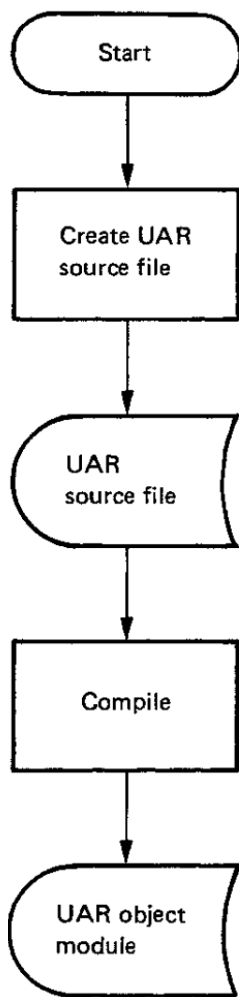
Figure 1.2, "Source Application Development Cycle" shows the steps in the development of the main program of an FMS application.

**Figure 1.2. Source Application Development Cycle****Write User Action Routines**

Using a text editor, write any user action routines that your application uses. UARs can exist in files by themselves or in the same file as the main program. Typically, application-specific UARs exist in the same file as the main program. General-purpose UARs are often compiled separately and are kept in object libraries.

Note that if you use UARs in your application, you must create a UAR vector module. Vector modules are described in *Chapter 7, "Programming Features"*.

*Figure 1.3, "UAR Development Cycle" shows the steps in the development of UARs.*

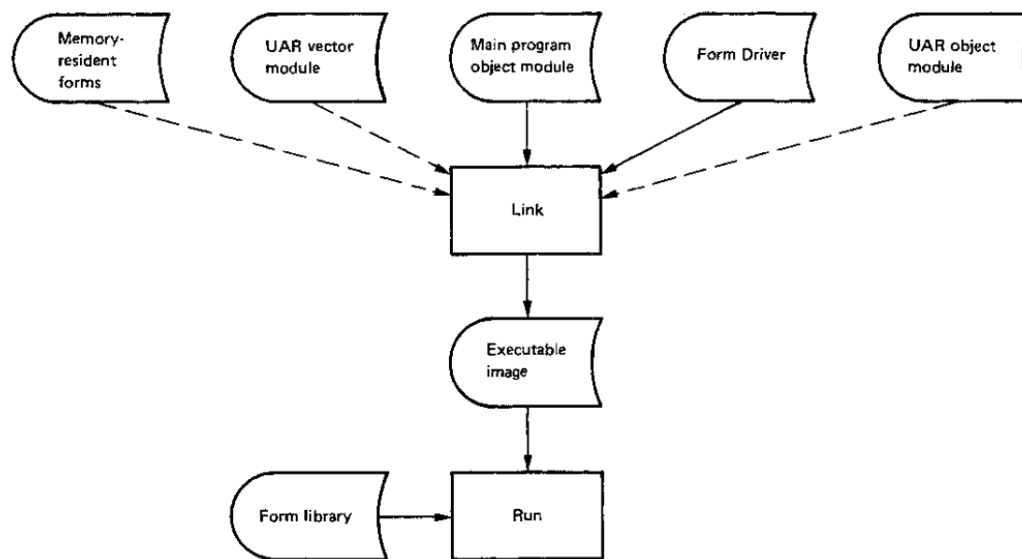
**Figure 1.3. UAR Development Cycle****Link the Application**

You link the following modules:

- Main program object module
- User action routine object module
- User action routine vector module
- Any memory-resident forms
- Form Driver

*Figure 1.4, "Linking of an FMS Application" is a flowchart that shows the linking of an FMS application.*



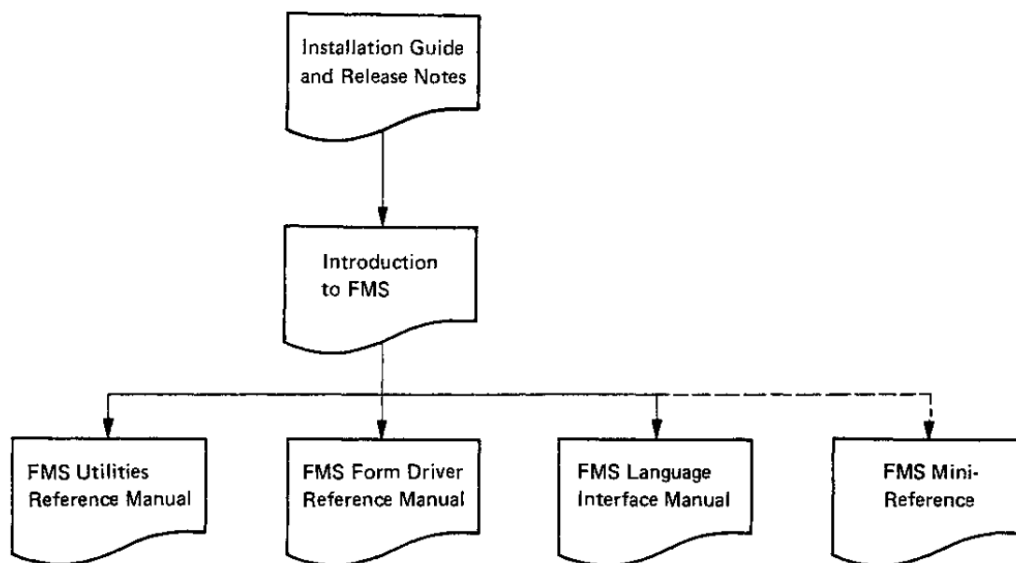
**Figure 1.4. Linking of an FMS Application**

## 1.3. Reading Path for the FMS Document Set

The OpenVMS FMS document set includes the following manuals:

- *VSI OpenVMS FMS Installation Guide*
- *VSI FMS for OpenVMS Systems Mini-Reference*
- *Introduction to VSI FMS*
- *VSI FMS Utilities Reference Manual*
- *VSI FMS Form Driver Reference Manual*
- *VSI FMS Language Interface Manual*

Figure 1.5, "FMS Document Reading Path" shows the reading path for this manual set. The manuals are described below.

**Figure 1.5. FMS Document Reading Path**

*VSI OpenVMS FMS Installation Guide* contains installation procedures for FMS on VAXNMS, a description of the Installation Verification Procedure, and information not included elsewhere in the document set. This manual describes the Form Upgrade Utility, which converts Version 1 forms and libraries to Version 2 forms and libraries.

This manual is intended for the person responsible for installing FMS and for upgrading any applications to Version 2.

*VSI FMS for OpenVMS Systems Mini-Reference* provides concise reference material for the contents of the other manuals. This manual also contains a diagram of the Form Editor keypad and a synopsis of compiling, linking, and running FMS applications.

This manual is intended for all users of FMS.

*Introduction to VSI FMS* introduces Version 2 of FMS and its utilities. The reader gains experience in creating forms, writing an FMS application, and using the Form Librarian, the Form Editor, and the Form Driver. This manual also contains a glossary of FMS terms.

This manual is intended for all users of FMS.

*VSI FMS Utilities Reference Manual* describes the following FMS utilities:

- Form Editor
- Form Language Translator
- Form Librarian
- Form Application Aids
- Form Tester

The manual includes examples of how to use these utilities.

This manual is intended for the application programmer and the person responsible for creating and maintaining forms and libraries.

*VSI FMS Form Driver Reference Manual* describes the function of the Form Driver and the Form Driver calls and explains how to use them. Programming techniques are discussed to help you with your FMS applications.

This manual is intended for the application programmer.

*VSI FMS Language Interface Manual* describes the FMS interface to the languages that FMS supports. A chapter is provided for each language that FMS supports, with examples in that language taken from the Sample Application. The Sample Application presented at the end of each chapter is written in the language of that chapter.

This manual is intended for the application programmer.



# Chapter 2. Running the Sample Application Program

As you read this chapter, you will:

- Run the Sample Application program, a checking account program
- See what the FMS features allow you to do
- Get a feel for being an operator as you work with FMS forms

Before continuing, check with your system manager to make sure that the FMS distribution kit has been installed on your system.

## 2.1. Setting Up Your Terminal

You can run FMS programs on any VT100 or VT100-compatible terminal. Before running the Sample Application, do the following:

1. Type the following command at the terminal on which you plan to run the Form Editor:

```
$SET TERMINAL/INQUIRE
```

In response, the OpenVMS system identifies the terminal you are using.

2. To see what the operating system knows about your terminal, type the following command:

```
$SHOW TERMINAL
```

In response, the system displays a list of your terminal's characteristics. Check to see that the list includes either the ANSI-CRT or the VT52 characteristic.

3. Check to see if the list obtained from the SHOW TERMINAL command includes the advanced video characteristic. If so indicated, your terminal has the advanced video option (AVO). If your terminal does not have AVO, the Sample Application as it appears on your screen will differ from what is shown in this manual. Specifically, a terminal without AVO:

- Can show reverse video or underlining, but not both
- Can display a maximum of 14 lines when set to a 132-column screen width

Terminals with AVO can display the blink and bold attributes and 24 lines when set to a 132-column screen width.

## 2.2. Starting the Sample Application

To start the Sample Application, enter the following commands:

```
$ RUN FMS$EXAMPLES:SAMP
```

The Sample Application displays the following image:

**Welcome to the FMS V2  
Sample Application Program (SAMP)**

**YOUR PERSONAL CHECKING ACCOUNT**

For instructions, press HELP (the PF2 key).  
To continue, press RETURN.

What you see is a form. This form identifies the application. You are told to press the PF2 key for on-line help and the RETURN key to continue. The HELP key is on the keypad to the right of the keyboard.

Press HELP. The following image appears.

**Help for the FMS V2 Sample Application Program**

The FMS Sample Application program (SAMP) serves two purposes:

1. It tests the Form Driver and is part of the Installation Verification Procedure.
2. It shows how to use FMS. The Sample Application is available in each language supported by FMS, and the documentation cites many examples that are from SAMP.

The application does not claim to show the best way of doing everything. Rather, it shows ways that things can be done with FMS.

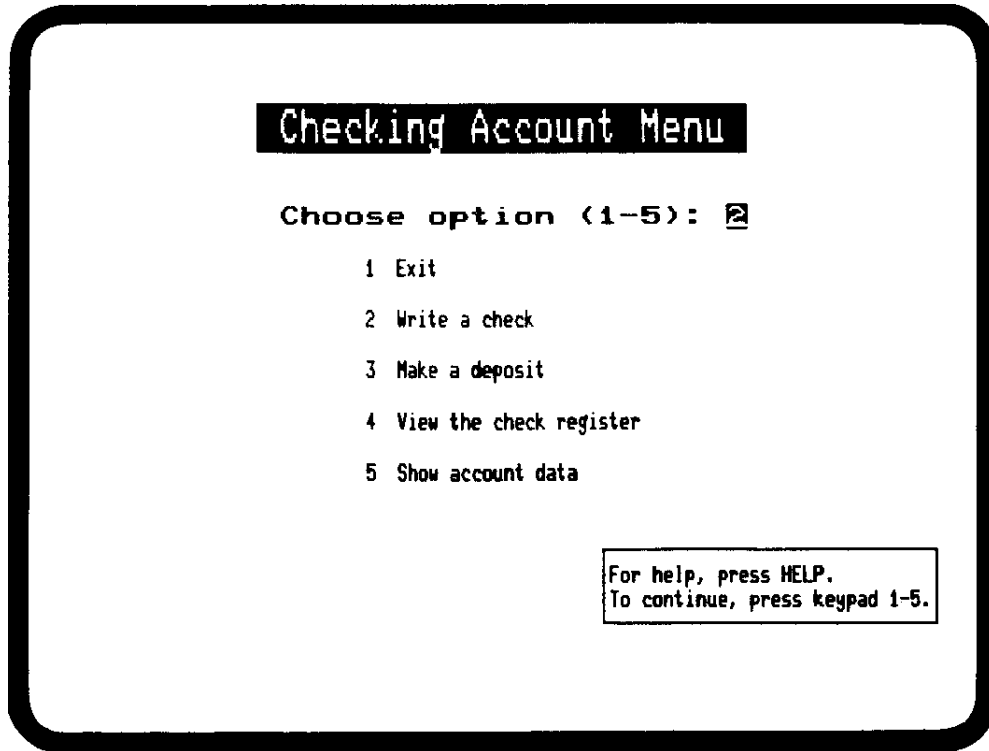
As you run the rest of SAMP, you can get help by pressing the PF2 key, which will be referred to as the HELP key. Repeated pressing of the key provides additional help until the message is displayed, "No help available." If you press HELP now, you will see an explanation of the keys used in FMS.

For more help, press HELP.  
To continue, press RETURN.

This image is a help form. This help form describes the Sample Application and tells how to get help while the program is running. If you press HELP again, you will see another form that gives general

instructions for entering data into forms and for moving the cursor from field to field. This second help form also tells you that you can press keypad period at any time to abort any procedure. If you press **HELP** again, a message appears at the bottom of the screen, saying that no more help is available. If you press **HELP** again, the first help form you saw appears. An FMS form can have any number of help forms associated with it. You will learn how to create help forms in *Chapter 3, "Creating Forms"*.

After you have seen all the help available with this form, press **RETURN**. The first form you saw, entitled **Welcome to the FMS V2 Sample Application Program**, will appear again. Press **RETURN** again. The following form appears.



```

Checking Account Menu

Choose option (1-5): 2

1 Exit
2 Write a check
3 Make a deposit
4 View the check register
5 Show account data

For help, press HELP.
To continue, press keypad 1-5.
```

This form is a menu, a form from which you can select actions that the application can perform. The title of this form, **Checking Account Menu**, is in double-size characters and in reverse video. The second line of this form, **Choose option (1-5):**, is in double-wide characters.

The cursor is located next to the caption *Choose option (1-5):*. The cursor is located in a field. In an FMS form, fields are used to display data or to accept and display data input by an operator. In the field shown above, the number 2 already appears. This number is the default for this field. If you do not specify a number, FMS will choose option 2.

Press **HELP**. The following message appears at the bottom of the screen:

```
Enter one of the numbers 1, 2, 3, 4, or 5
```

This help line gives you assistance for the field in which the cursor is positioned.

Press **HELP** again. The help form that appears describes the menu's options. If you press **HELP** again, the second help form associated with the first form you saw appears. After you have examined the help forms associated with the menu, return to the menu by pressing **RETURN**.

Before entering an option number, type an alphabetic character. When you do, the terminal beeps, and a message is displayed on the bottom line of the screen. As the Sample Application displays a form, the Form Driver checks to see if input to a field is valid. If you try to enter an alphabetic character into a field that can accept only numbers, the Form Driver indicates that the input is invalid by signaling you with a beep and by displaying the following message:

Numeric required

Now type a 2-digit number. When you do, a message is displayed on the bottom line, indicating that the field is full. The Form Driver can also limit the number of characters that can be entered in a field.

Select option 2, Write a check, by pressing keypad 2. You can also press keyboard 2, followed by RETURN, to select option 2. If you receive the Fieldfull message, press DELETE; then type 2, followed by RETURN. The following form appears:

The name and address on this form, in addition to other account data, were provided by an account data file supplied with the Sample Application. The cursor is located next to the caption *Pay to*. Before filling in this field, press HELP to see what help is available.

The help form associated with this form tells you that you can fill in fields and that you can move the cursor forward to the next field by pressing TAB and backward to a previous field by pressing BACKSPACE. The help form also tells you that when you are satisfied with the entries you have made, you can press RETURN to enter the check into the check register. You can abort the check-writing process by pressing keypad period, which returns you to the menu.

Note that the help form specifies that the keypad period key has been assigned a special function in the Sample Application. Assigning functions to keypad keys is a feature available to your FMS applications.

The *Pay to* field has the Response Required attribute. Before you finish filling in this field, press LINEFEED. Doing so deletes the contents of the field. You can move the cursor left or right within the field by pressing CHARBCK or CHARFWD, respectively. After filling in the *Pay to* field, press RETURN. The Form Driver signals you by beeping and displays the following message on the bottom line of the screen:

Input required

The cursor is now positioned in the next field, *Amount*. If you press RETURN, the terminal beeps, and a message is displayed at the bottom of the screen, indicating that input is required.



Note that the cursor is positioned to the right of the *Amount* field. As you enter numbers in this field, they appear to the left of the cursor and are pushed to the left as you continue to type numbers. This field is right justified; that is, the operator can enter numbers as in a calculator or an electronic cash register.

Now enter an alphabetic character into this field. The following message appears at the bottom of the screen:

```
Numeric required
```

Only numbers can be entered in this field. Fill in the next field, *Memo*, and press RETURN. Note that the current balance figure, shown below the check, is updated and that the following message appears at the bottom of the screen:

```
Your check has been written and sent to the payee's account,  
To return to menu, press Keypad period.  
To print check into file "SAMPCH,DAT", press Keypad zero.  
To write another check, press Return
```

Now press RETURN or ENTER. Make out another check. This time, however, make it out in an amount greater than the current balance. When you press RETURN, a message appears at the bottom of the screen, saying that you attempted to write a check that exceeds the current balance. Enter an appropriate amount and press RETURN.

You can write another check if you wish, or you can return to the main menu by pressing keypad period.

When you return to the menu, select option 3, Make a deposit, by pressing keypad 3. The following form appears on the screen:

MAKE A DEPOSIT

Date: 17-SEP-82

Current Balance	\$ 361.30
Deposit	<u>\$0000.00</u>
New Balance	\$ .

Memo: \_\_\_\_\_

Take a moment to examine the form and read the available help. Now enter a deposit. Note that the cursor is positioned on the decimal point in this field. As you enter numbers, they are pushed to the left of the decimal point. This field has the Fixed Decimal attribute. If you type a period and then enter more numbers, they appear to the right of the decimal point.

After you fill in the memo and press **RETURN**, a message appears, saying that the deposit has been made.

When you are through examining the Make a Deposit form, return to the menu and select option 4, View the check register, by pressing keypad 4. The following form appears on the screen:

CHECK REGISTER - THE ACCOUNT HISTORY					
Chk. No.	Date	Check Payee or Deposit Memo	Deposit Amount	Check Amount	New Balance
	15-MAR-82	Interest on National Coal bond	500.00	.	500.00
1	15-MAR-82	Jack Dewar	.	10.00	490.00
2	30-JUN-82	Louise Phipps	.	20.00	470.00
3	14-JUL-82	Townsend Fabrics	.	250.00	220.00
4	30-JUL-82	Channel 42	.	50.00	170.00
	31-AUG-82	Paycheck	300.00	.	470.00

This Session: Starting Balance: \$ 361.30  
 Total Deposits: \$ . 0  
 Total Checks: \$ . 0  
 Current Balance: \$ 361.30

To scroll through the check register, press UPARROW or DOWNARROW.  
 To return to the menu, press RETURN.

This form shows the FMS scrolling feature. Scrolling enables you to display more lines than can fit on one screen. Look at the list in the check register. Each item in the list records information on a check written or a deposit made. Note the message at the bottom of the form:

To scroll through the check register, Press UPARROW and DOWNARROW.  
 To return to the menu, press RETURN.

Now press DOWNLINE to scroll forward. The list of register items scrolls up one item. That is, the item that was at the top of the list has disappeared, and a new item has appeared at the bottom of the list.

Press DOWNLINE several times to scroll to the last item in the check register. When the last item is reached, the following message appears on the bottom of the screen:

Last line of resister

Press UPLINE to get to the top of the check register. When you reach the top, a message appears, saying that the top of the check register has been reached. (*Chapter 7, "Programming Features"* describes how scrolling was implemented in the Sample Application.)

When you are through examining the check register, return to the menu and select option 5, Show Account Data, by pressing keypad 5. The following form appears on the screen:

## ACCOUNT DATA

ACCOUNT NUMBER: 888
Account opened: 15-MAR-81

NAME Last: Smith First: Katherine Middle: M.

ADDRESS Street: 1 Hog Hill Rd.  
City: OWNERSD State: AK Zip: 99999

PHONE Home: (800)555-1212 Business: (800)555-1111

Enter secret password to change the account data:

To record new account data and return to the menu, press RETURN.  
To return to the menu without changing the data, press keypad period.

This form requests a password to change the account data. Press HELP. The help line provides the password. Before entering the password, press HELP again to see a description of this form. The Sample Application requires that you enter the password in uppercase letters, as indicated in the help line.

You can alter any of the data in the fields of this form, but these modifications are not permanent, as explained in the help form. When you run the Sample Application again, this form will look the same as when you first ran the program. This is how SAMP was designed; FMS does not restrict you from making permanent changes to any data base.

The password is an example of the FMS No Echo attribute. You can include this feature in your applications to provide a measure of protection. Whenever an operator enters data into a field with the No Echo attribute, the data goes to the application program, but does not appear on the screen.

When you are through examining the Account Data form, you can return to the menu and exit by pressing keypad 1, or you can reexamine some of the forms.

## 2.3. Printing SAMPCH.DAT

When you ran the Sample Application and wrote a check, you had the option of sending a check to the data file SAMPCH.DAT for subsequent printing. In this section, you will get a hard-copy listing of that data file. When you specify that you want a check sent to SAMPCH.DAT, the Sample Application sends SAMPCH.DAT to your main directory.

To obtain a hard-copy listing of SAMPCH.DAT, type the following command:

```
$ PRINT SAMPCH,DAT
```

The following is a sample listing of SAMPCH.DAT:

```
+-----+
| Katherine M. Smith                      Number 8      |
| 1 Hog Hill Rd.                         |
+-----+
```

Townsend, AK 99999	Date 28-SEP-82	
Pay to L. Martin	Amount: \$**45.95	
Memo Roofing supplies		
FIRST NATIONAL BANK	Account 532	
+-----+		

# Chapter 3. Creating Forms

As you read this chapter, you will:

- Learn how to set your terminal to run the Form Editor
- Examine two forms from the Sample Application
- Create two forms that appear in the Sample Application
- Create a help form

## 3.1. Setting Up Your Terminal

You can use the Form Editor on a VT100 or a VT100-compatible terminal. Before running the Form Editor, do the following:

1. Type the following command:

```
$ SET TERMINAL/INQUIRE
```

In response, the OpenVMS system identifies the terminal you are using.

2. To see what the operating system knows about your terminal, type the following command:

```
$ SHOW TERMINAL
```

In response, the system displays a list of your terminal's characteristics. Check to see that the list includes the DEC-CRT characteristic.

3. Check to see if the list obtained from the SHOW TERMINAL command includes the advanced video characteristic. If so indicated, your terminal has the advanced video option (AVO). If your terminal does not have AVO, the Form Editor as it appears on your screen will differ from what is shown in this manual. Specifically, a terminal without AVO:

- Can show reverse video or underlining, but not both
- Display a maximum of 14 lines when set to a 132-column screen width

Terminals with AVO can display the blink and bold attributes and 24 lines when set to a 132-column screen width.

## 3.2. Examining Two Forms from the Sample Application

In *Chapter 2, "Running the Sample Application Program"*, you ran the Sample Application from the FMS distribution kit. In this chapter, you will look more closely at two of the forms in that application program.

First, rerun part of the Sample Application so that you can see the two forms again. As in *Chapter 2, "Running the Sample Application Program"*, run the Sample Application by typing the following commands:

```
$RUN FMS$ENAMPLES:SAMP
```

You will first examine the forms and then create them yourself.

### 3.2.1. MENU Form

After you see the form entitled Welcome to the FMS V2 Sample Application Program, press RETURN. The menu, entitled Checking Account Menu, appears. We will call this form MENU.

In the MENU form shown below, the numbers above and to the left of the form are line and column indicators for the exact positions of fields and the background text.

	10	20	30	40	50	60	70	80
1	<div style="border: 2px solid black; border-radius: 15px; padding: 10px; text-align: center;"> <p><b>Ⓐ → Checking Account Menu</b></p> <p><b>Choose option (1-5): 2 ← Ⓑ</b></p> <div style="display: flex; align-items: center; justify-content: center;"> <div style="font-size: 3em; margin-right: 10px;">{</div> <div style="text-align: left;"> <p>1 Exit</p> <p>2 Write a check</p> <p>Ⓒ 3 Make a deposit</p> <p>4 View the check register</p> <p>5 Show account data</p> </div> </div> <div style="border: 1px solid black; padding: 5px; margin-top: 20px; text-align: center;">             For help, press HELP.              To continue, press keypad 1-5.           </div> </div>							
12								
20								
22								

MENU has the following design features:

1. The form appears with white background and dark text. The form appears in reverse video.
2. The title, **A**, is centered at the top and has double-size characters. The title is in reverse video.
3. This form's only field, **B**, appears two lines below the title and is underlined. When this form appears on the screen, a 2 is already in the field; the 2 is the default value for this field. This field, along with its caption, *Choose Option (1-5)*: is centered on line 7 and has double-wide characters. The caption is background text that identifies the field; that is, the field can be referred to as the *Choose option* field.
4. More background text, **C**, starts in column 27 in lines 9, 11, 13, 15, and
5. The boxed background text in the bottom right of the form gives instructions to the operator.

Consider the purpose of MENU. The operator uses this form as a menu for selecting an action listed in the background, **C**. The operator types the number of the action to be taken. Only numbers in the range 1 to 5 are accepted. If the operator types a number outside that range or a non-numeric character, the Form Driver does not accept it, beeping the terminal and displaying an error message at the bottom of the screen.

Press HELP. A message appears at the bottom of the screen, **D**. When you create this form, you will provide the help text that appears at the bottom of this form.

### 3.2.2. DEPOSIT Form

After you have examined MENU on the screen, press keypad 3. The form entitled Make a Deposit appears. We will call this form DEPOSIT. The DEPOSIT form is shown below.

The screenshot shows a terminal window with a thick black border. Inside, the form is titled 'A MAKE A DEPOSIT'. Below the title, there are five fields, each with a number 1 through 5 to its right, indicating their positions. Field 1 is 'Date: 24-SEP-82'. Field 2 is 'Current Balance \$ 361.30'. Field 3 is 'Deposit \$0000.00'. Field 4 is 'New Balance \$ .'. Field 5 is 'Memo: ' followed by a long underline.

```

      A MAKE A DEPOSIT
                                Date: 24-SEP-82 ← 1
Current Balance  $ 361.30 ← 2
Deposit         $0000.00 ← 3
New Balance     $ . ← 4
Memo: _____ ← 5
  
```

DEPOSIT has the following design features:

1. The form appears with a white screen background and dark text, as in MENU.
2. The title, **A**, is centered at the top of the screen and is in uppercase bold. The title starts in column 32.
3. Field 1, with its caption, is to the right on the form. The caption *Date*: is in column 49 on line 3.
4. Fields 2, 3, and 4, with their captions, appear below the title. These fields begin in column 42 on lines 5, 7, and 9. The captions *Current Balance*, *Deposit*, and *New Balance* begin in column 22 on lines 5, 7, and 9.
5. Field 5 is underlined. The field's caption, *Memo*, begins in line 12, column 22. Field 5 spans columns 28 to 62, occupying 35 character spaces.

Although DEPOSIT has five fields, the operator can enter data only into fields 3 and 5. The remaining three fields are display only. As the Sample Application runs, it displays data in those three fields. However, when you lay out the fields, you are not concerned whether fields are display only.

Note the attributes of each field in DEPOSIT:

1. Field 1 is a date field. When the Sample Application runs, the system date is automatically inserted.
2. Field 2 has space for six digits: four to the left of the decimal point and two to the right. The decimal point is a field marker. Because this field is display only, it has no corresponding help text.

3. Field 3 is arranged like field 2; the operator can enter only numbers in this field. Note that this field is fixed decimal. That is, as the operator fills in this field, the numbers are placed from right to left. After the operator types a decimal point, the remaining numbers are automatically inserted to the right of the decimal point. The help text for this field is "Enter amount of deposit." If the operator fails to enter an amount in this field and attempts to tab to the next field, the following message appears at the bottom of the screen:

Input required

Thus, the *Deposit* field has the Response Required attribute.

4. Field 4 has space for six digits. Because this field is display only, it has no corresponding help text.
5. Field 5 is underlined. Any printable character can be entered in this memo field. The help text supplied is "Enter origin of deposit."

You use the Form Editor to assign these special characteristics to each field in DEPOSIT. The procedure for doing this is provided later in this chapter.

After you have examined DEPOSIT, press RETURN or ENTER to get back to the main menu. You can either exit from the Sample Application or look again at MENU or DEPOSIT.

## 3.3. Creating MENU and DEPOSIT

In this section, you will create MENU and DEPOSIT. You will do the following for each form:

- Assign a name
- Lay out the background text and fields
- Assign special attributes to each field
- Test to verify that the visual characteristics work properly and that the fields accept data as intended

### Using the Form Editor

The Form Editor, like the Sample Application, is a program that uses forms for a variety of purposes. When you start the Form Editor, a menu appears on the screen, prompting you for a specific action to take, such as assigning attributes to a form you wish to create or laying out the form. This chapter does not explain how to use every feature of the Form Editor; rather, it tells how to use those features you need to create the two forms from the Sample Application. For complete details on assigning attributes, see the *VSI FMS Utilities Reference Manual*

As you use the Form Editor, you make extensive use of the keypad to the right of the terminal's keyboard and the arrow keys above the keyboard. You use the keypad for the following operations:

1. Moving the cursor. For example, you can move the cursor up or down any number of lines and from side to side. Using two keypad keys, you can also instantly move the cursor to the top left of the screen or to the bottom right.
2. Deleting characters. You can delete individual characters, entire character strings, and whole lines.
3. Changing editing modes. The modes are described later.
4. Creating special video characteristics. You can make any character or character string have special video qualities, such as blinking, holding, or underlining. You can make characters double size or double wide, and you can draw lines and boxes on the screen.



5. Inserting any amount of text into a special buffer and "pasting" it elsewhere on the screen.
6. Centering text on a line.

## Note


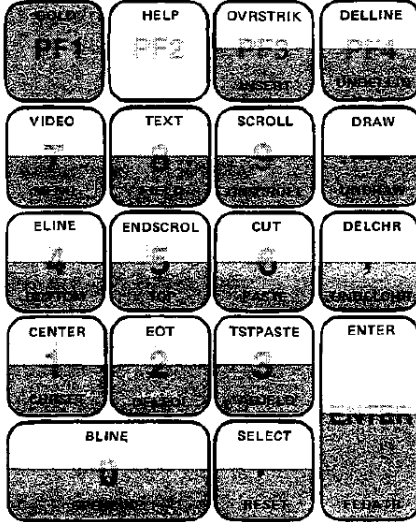
You should have the keypad stickup for the Form Editor, which is provided with the FMS distribution kit, before continuing with this chapter.

Figure 3.1, "Form Editor Keys" shows the Form Editor keys.

Each keypad key has one or two uses. For example, the key in the top right is labeled DELLINE and UNDELLINE. If you press this key while laying out a form, the Form Editor deletes all the characters on the line the cursor is in. You have performed the function shown at the top of the key.

To perform the functions shown at the bottom of the keys, you first press the PF1, or GOLD, key and then the key labeled with the operation. Pressing GOLD and any other key having two labels executes the function named at the bottom of the second key. For example, if you press GOLD and then the DELLINE/UNDELLINE key, you restore the last line deleted by the DELLINE function.

**Figure 3.1. Form Editor Keys**

					
<b>ALL PHASES</b>					
RETURN	Terminates display of current form (in Layout, moves cursor to next line)				
DELETE	Deletes previous character				
LINEFEED	Deletes field contents (in Layout, moves cursor down one line in same column)				
TAB	Moves cursor to next field (in Layout, moves cursor to next fixed tab stop)				
BACKSPACE	Moves cursor to previous field (in Layout, moves cursor to previous character position)				
GOLD Q	Reverses current error signaling mode				
GOLD R	Restores original field values (except in Layout)				
CTRL/R	Refreshes the screen				
<b>LAYOUT PHASE ONLY</b>					
GOLD n	Repeats a key or operation <i>n</i> times				
GOLD D	Creates a date field				
GOLD T	Creates a time field				
GOLD S	Makes current line double size				
GOLD W	Makes current line double wide				
CTRL/U	Deletes to beginning of line				
		<b>ORDER PHASE ONLY</b>			
		GOLD C Restores conventional field access order			
		<b>TEST PHASE ONLY</b>			
		GOLD ↑ Exits to previous field from scrolled area			
		GOLD ↓ Exits to next field from scrolled area			

When the instructions in this chapter tell you to press a specific key – for example, DELLINE – press the key labeled DELLINE. When you are instructed to press GOLD UNDELLINE, press the GOLD key and then the UNDELLINE key.

### 3.3.1. Creating MENU

MENU has only one field; in the Sample Application, the field accepts only a number from 1 to 5. When you create MENU, however, this field will accept any integer in the range 0 to 9. To restrict the integers that will be accepted requires programming procedures that are described later in this manual.

The first step in creating MENU is to type the following:

```
$FMS/EDIT
```

FMS responds by displaying the following prompt:

```
_File:
```

Type the name of the form you wish to create, which initially exists as a file:

```
_File: MENU
```

MENU is the name of the file containing the form. By default, FMS assigns a :FRM file type.

As an alternative to the commands shown above, you can type the following command on one line:

```
$ FMS/EDJT MENU
```

FMS responds by clearing the screen and then displaying the Form Editor menu, shown in *Figure 3.2, "Form Editor Menu"*.

**Figure 3.2. Form Editor Menu**

```
Form Editor Menu

Phase Choice: _____

Form  Assign form attributes
Layout Create or modify a form
Assign Assign field attributes
Data  Enter Named Data items
Order Modify field access order
Test  Test the form with the Form Driver
Exit  End this editor session

Form Name: MENU
Input File: New form being created
```

You use this menu to select an action to perform. You type in the name of the phase you wish to enter.

First, you assign characteristics that apply to the entire form you are about to create. The Form phase accomplishes this. To enter the Form phase, type FORM; then press RETURN or ENTER.

### 3.3.1.1. Assigning Form Attributes: The Form Phase

You are now in the Form phase. The Form Attributes questionnaire appears on the screen (see *Figure 3.3, "Form Attributes Questionnaire"*). This form contains several fields, but now you need be concerned only with the screen background field.

**Figure 3.3. Form Attributes Questionnaire**

```

Assign Form Attributes

Form Name:  MENU
Help Form Name:

Screen Background: 3
  1. As Is
  2. Black
  3. White

Screen Width: 1
  1. As Is
  2. 80 Columns
  3. 132 Columns

Screen Character Set: 1
  1. As Is
  2. US
  3. UK
  4. RULE
  5. SET1
  6. SET2

Screen Area to Clear
  First Line 1
  Last Line 23

Field Highlighting
  X No Highlighting
  _ Blink
  _ Bold
  _ Reverse
  _ Underline

Do you want to specify user action routines for this form? (Y/N) N
Do you want to assign initial field attributes? (Y/N) N

```

When the Form Editor first displays this form on the screen, note where the cursor is positioned. The Form Editor has already assigned the name MENU to this form; to keep this default form name, press TAB.

Press TAB again to move the cursor to the field captioned *Screen Back ground*:. If the cursor goes past this field, use the BACKSPACE key to back up. Type 3 to indicate that MENU is to have a white background. You are now finished with your work in the Form phase.

To get back to the menu, press GOLD MENU. The Form Editor responds by clearing the screen and displaying the menu. You are now ready to enter the Layout phase.

### 3.3.1.2. Laying Out the Form: The Layout Phase

In the Layout phase, you type in the background text and fields that make up a form. To enter the Layout phase, type LAYOUT and then press RETURN.

As you enter the Layout phase, the screen is cleared, and the Layout phase status line appears at the bottom of the screen. This line, shown in *Figure 3.4, "Layout Phase Status Line"*, provides information about the Form Editor as it is operating. *Table 3.1, "Layout Phase Status Line Information"* describes the information given in the status line.

**Figure 3.4. Layout Phase Status Line**

```

Cursor TXT NOR Line 1 Column 1 Modes TXT OVS

```

**Table 3.1. Layout Phase Status Line Information**

Item	Meaning
<b>Cursor</b>	
TXT or FLD	The character type at the cursor position can be either text (TXT) or field (FLD).
NOR or SCR	Indicates whether the line on which the cursor is positioned is normal (NOR) or scrolled (SCR).
LINE (1-23)	Indicates the line number.
COLUMN (1-132)	Indicates the column.
<b>Modes</b>	
TXT or FLD	Indicates the mode that selects whether background text (TXT) or fields (FLD) are entered.
OVS or INS	Indicates how characters are entered on the screen. Character entry mode can be either Overstrike (OVS) or Insert (INS).
<b>Field Name</b>	Displays the name of the current field. If you do not assign a field name, the Form Editor provides a default field name of F\$nnnn.

**Type the Background Text for MENU**

Type in the background text for MENU. Position each character in the exact column and line as described.

Start with the title. Noting that the title begins on line 2, move the cursor to the beginning of line 2. To move the cursor down, you can press either DOWNLINE, found at the top right of your terminal's keyboard, or RETURN. The line field in the status line at the bottom of the screen is updated as you move the cursor. Referring to this status data, you can always keep track of the exact line and column in which the cursor is located.

```
Checking Account Menu

Choose option (1-5): 2

1 Exit
2 Write a check
3 Make a deposit
4 View the check register
5 Show account data

For help, press HELP.
To continue, press keypad 1-5.
```

As you follow the procedures in this exercise, you are instructed to make deliberate errors. You will then be directed to go back over your work and to make the necessary corrections. In this way, you get an opportunity to learn more about the Form Editor.

When the cursor is at the beginning of line 2, press GOLD S to indicate that you want that line to have double-size characters. As you press GOLD S, the cursor instantly doubles in size, and the line counter at the bottom of the screen indicates that you are in line 3. The double-size characters you are about to type will occupy lines 2 and 3. By checking the cursor size, you can always determine whether the line contains normal-size characters. Press GOLD S again; the cursor returns to its original size. Pressing GOLD S yet again returns you to double-size mode, as indicated by the size of the cursor. In double-size characters, type the title as shown here:

```
Checking Account Menu
```

Note the deliberate misspelling of Account. Later you will go back and correct this error. If you make any other errors as you type, use the DELETE key to make corrections. Before proceeding, press GOLD S to change character size.

### Center the Title

Next, use the Form Editor CENTER function to center the title. To use the CENTER function, make sure that the cursor is located on line 3, the line you wish to center. Press CENTER.

After you have centered the title, move the cursor down to the beginning of line 7. You can move the cursor to the beginning of line 7 in two different ways:

1. Press RETURN four times.
2. Press DOWNLINE four times; then press CHARBCK until the cursor is at the beginning of the line.

When the cursor is at the beginning of line 7, press GOLD W to make this line have double-wide characters. As with GOLD S, you can change back and forth between character size each time you press

GOLD W. You can also check the cursor size to determine which character size mode you are in. Type in the following text in double-wide characters:

Choose Option (1-4):

Now place the cursor at the beginning of line 7 and press CENTER to center the text you just typed.

Next, move the cursor down to line 9, column 27, and type in the following text:

1 Exit

Type in the remaining options for MENU. Then move the cursor down to line 21, column 50, to type in the instructions:

For help, press HELP.

To continue, press keypad 1-5.

### **Draw a Box**

You now want to enclose the text in a box. Move the cursor to line 20, column 49. This character position is immediately to the left of and above the instructions. To use the DRAW function to create a box, follow the procedure given here:

1. With the cursor in line 20, column 49, press SELECT.
2. Press CHARFWD until the cursor is in column 80.
3. Press DOWNLINE to move the cursor down three lines to line 23. Note that the entire area of instructions is in reverse video. The reverse video indicates the screen area that is in a buffer known as the select range – the screen area that is subject to a Form Editor function you choose.
4. Press DRAW.

If you make an error in the placement of the lines, you can put the lines back in a select range and press GOLD UNDRAW.

---

### **Note**

When you create a select range, FMS works more quickly if you move the cursor horizontally before moving it vertically.

---

When you have typed in all the background text for MENU, you are ready to create the field.

### **Create a Field for MENU**

The one field in MENU follows the caption *Choose Option (1-5):*. Move the cursor a space beyond the colon. As the cursor enters line 7, it instantly changes to double-wide size.

With the cursor in place, you are ready to create the field. Put the Form Editor into Field mode by pressing GOLD FIELD. The Form Editor indicates that it is in Field mode by displaying FLD in the status line at the bottom of the screen in the Modes field.

Type the following:

9

The character 9 is a field-validation character. This character is a symbol that tells the Form Driver to accept only an integer in this field during run time. If the operator attempts to enter a non-numeric

character in this field, the Form Driver will reject it, causing the terminal to beep and the screen to display a message.

After you have entered the 9, return to Text mode by pressing TEXT. The Form Editor indicates that it is in Text mode by displaying TXT in the status line at the bottom of the screen in the Modes field.

### Correct MENU

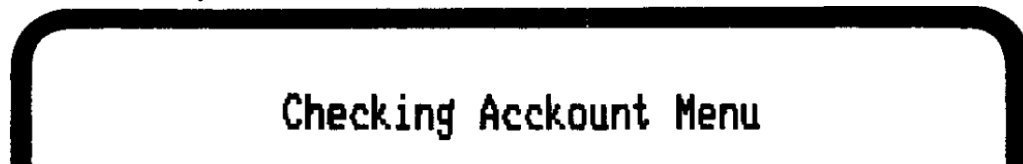
Now review your work. Go back and correct the spelling errors that you intentionally entered.

1. Correct the misspelling in the title.
2. Correct the caption for the *Choose option* field.

As you correct your form, you will be experimenting with two of the Form Editor's operating modes: Insert mode and Overstrike mode. These two operating modes refer to the way in which the Form Editor places characters on the screen.

- Insert Mode

To begin correcting MENU, press GOLD INSERT to put the Form Editor into Insert mode. Then press GOLD TOP to position the cursor at the top left of the screen. Next, use the CHARFWD and DOWNLINE keys to move the cursor to the *k* in **Acckount**. Press DELCHAR.



When you press DELCHAR, all the text to the right of the cursor moves over one column to the left to fill in the space vacated by the deleted *k*.

- Overstrike Mode

To correct the next error in MENU, the text that accompanies this form's field, press OVERSTRIKE to put the Form Editor into Overstrike mode. Move the cursor to 4 in *Choose option (1-4):*. You can move the cursor either by using the arrow keys or by pressing RETURN and CHARFWD until the cursor is in position. When the cursor is in position, type 5. The new number replaces 4 in the column. The rest of the characters on line 3 are unaffected.

### Assign Video Attributes for MENU

You are now ready to assign video attributes to your form. Video attributes – blink, bold, reverse, and underline – are assigned through a keypad key.

Two video attributes are in MENU:

1. The title, Checking Account Menu, is in reverse video.
2. The field, next to *Choose Option (1-5):*, is underlined.

To make the title bold, do the following:

1. Position the cursor in the character location immediately preceding the C in Checking.
2. Press SELECT.
3. Move the cursor to the character position immediately following the u in Menu.

4. Press VIDEO on the keypad. The Form Editor responds by displaying the VIDEO: prompt at the bottom of the screen.
5. Type BOLD in response to the prompt:  
  
VIDEO: BOLD
6. Press ENTER. The Form Editor responds by holding the title and repeating the VIDEO: prompt at the bottom of the screen.
7. Type SAVE to save the video attributes just assigned:  
  
VIDEO: SAVE
8. Press ENTER.

To underline the field, do the following:

1. Move the cursor to the 9 in the field.
2. Press VIDEO.
3. Type UNDERLINE in response to the prompt at the bottom of the screen:  
  
VIDEO: UNDERLINE
4. Press ENTER. An underline appears under the 9, and the VIDEO: prompt appears again at the bottom of the screen.
5. Type SAVE to save the video attributes just assigned:  
  
VIDEO: SAVE
6. Press ENTER.

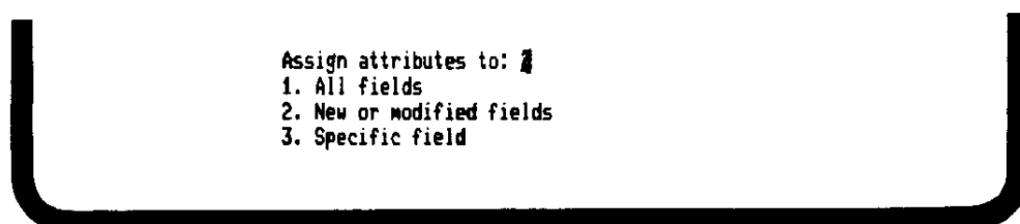
You have now completed your work in the Layout phase. The next step is to assign field attributes. For MENU, the only field attributes you assign are Field Name and Response Required. You do this in the Assign phase.

To exit from the Layout phase and return to the Form Editor menu, press GOLD MENU. Pressing GOLD MENU at any time during Layout returns you to the menu.

### 3.3.1.3. Assigning Field Attributes: The Assign Phase

You are now ready to assign other attributes to the field in MENU. To enter the Assign phase, type ASSIGN and then press RETURN. The Form Editor responds by displaying a question below the menu (see *Figure 3.5, "First Assign Phase Question"*).

**Figure 3.5. First Assign Phase Question**





Press RETURN if you want to assign attributes to a new field, the default choice. The screen is cleared, and MENU appears, along with a questionnaire at the bottom, as shown in *Figure 3.6, "Second Assign Phase Questionnaire"*.

**Figure 3.6. Second Assign Phase Questionnaire**

```

Checking Account Menu
Choose Option (1-5): 3

1 Exit
2 Write a check
3 Make a deposit

Assign Field Attributes
Field Name: F$0001 Index Value 1 of 1
- Autotab      - Right Justify  - Uppercase
- No Echo      - Fixed Decimal  - Must Fill
- Display Only - Zero Fill      - Response Required
-              - Zero Suppress  - Supervisor Only
Default Value:
Help Text:

```

When the Form Editor displays this questionnaire on the screen, the cursor is positioned at **A**, in the *Name* field. In the MENU form, the field is highlighted in reverse video to tell you which field is being assigned attributes. MENU has only one field, but if there were more than one field, you would need to know which field you were working on.

Note that F\$0001 appears as the field name. This default field name is assigned if you do not specify a name.

Type **OPTION** for the field name. Then press **TAB** to move the cursor through the list to the blank captioned *Response Required*. Type **X**. Assigning this attribute means that when MENU is displayed during run time, the operator must enter a value in this field.

Move the cursor down to the field captioned *Default value*:. Type **2**. The number you type in this field is the value that the Form Driver displays when the form first appears on the screen. That is, when the Sample Application displays MENU on the screen, the default value is automatically displayed. If the operator does not supply another number in this field, the Form Driver automatically performs this action.

Press **TAB** to move the cursor to the field captioned *Help text*:. Here you enter the single-line help message that the operator can see by pressing **HELP** as the Sample Application is running. The help message for this field is:

Enter one of the numbers 1, 2, 3, 4, or 5

After you type in the help message, press **GOLD MENU** to return to the menu.

### 3.3.1.4. Testing a Form: The Test Phase

The Test phase lets you see how MENU would be displayed if it were being displayed by the Sample Application. Type TEST and then press RETURN to enter the Test phase. Check the following:

1. Make sure that the visual characteristics are as you intended. Check the positions of the title, the background text, and the field.
2. Press HELP. Check to make sure that the help message appears correctly.
3. Enter an alphabetic character. Check to make sure that the correct error message appears.

When you are satisfied that MENU is correct, press RETURN or ENTER to return to the menu.

### 3.3.1.5. Saving a Form: The Exit Phase

After you return to the menu, type EXIT and then press RETURN. The Form Editor responds by displaying the following:

```
Do you wish to save this form? Y
```

Press RETURN if you wish to save the form you just created. If you type N and then press RETURN, the Form Editor deletes the form you just created.

## 3.3.2. Creating DEPOSIT

You create DEPOSIT just as you created MENU. The following steps summarize the procedure:

1. Invoke the Form Editor and type DEPOSIT as the name of the form you wish to create.
2. When the menu appears on the screen, type FORM to enter the Form phase. In the Form phase, specify a white screen background. Return to the menu.
3. Type LAYOUT to enter the Layout phase.
  - In Layout, type in the background text, as planned.
  - Create the four fields.
  - Correct any errors, such as spelling, that may appear on the screen.
  - Assign video attributes to the background text and to the fields, using the VIDEO key.
  - Return to the menu.
4. Type ASSIGN to enter the Assign phase.
  - Assign field names, help text, and default values for each field.
  - Return to the menu.
5. Type TEST to enter the Test phase. Try out your form.
6. When you are satisfied with your form, type EXIT and indicate that you want to save your form.

Review the design for DEPOSIT. Note the special video characteristics.

```

      MAKE A DEPOSIT

                        Date: 24-SEP-82

Current Balance    $ 361.30
Deposit           $0000.00
New Balance       $ .

Memo: _____

```

To invoke the Form Editor, type the following:

```
$ FMS/EDIT DEPOSIT
```

When you press RETURN, the Form Editor clears the screen and displays the menu.

### 3.3.2.1. Assigning Form Attributes: The Form Phase

With the menu on the screen, type FORM and then press RETURN. The Form Editor responds by displaying the Form Attributes questionnaire. Press TAB to move the cursor to the *Screen Background* field. Type 3 to assign a white screen background to DEPOSIT. You are now finished with your work in the Form phase.

```

      Assign Form Attributes

Form Name:  DEPOSIT
Help Form Name:

Screen Background: 3
1. As Is
2. Black
3. White

Screen Width: 1
1. As Is
2. 80 Columns
3. 132 Columns

Screen Character Set: 1
1. As Is    4. RULE
2. US       5. SET1
3. UK       6. SET2

```

Press GOLD MENU to return to the menu.

### 3.3.2.2. Laying Out the Form: The Layout Phase

You are now ready to begin laying out DEPOSIT. Type LAYOUT to enter the Layout phase.

Type the Background Text for DEPOSIT

Type in the background text for DEPOSIT as shown in the SAMP form in *Section 3.3.2, "Creating DEPOSIT"*. Note that the dollar sign (\$) for the *Current Balance*, *Deposit*, and *New Balance* fields is background text. The dollar sign is not part of the field. Position the cursor in the column 49, where the *Date* field begins. Type in the following caption:

Date:

### Create a Date Field

When you have finished typing in the background text for DEPOSIT, you are ready to create the fields. The first field you create is the *Date* field.

With the cursor in column 55, press GOLD D. The Form Editor responds by displaying a list of date formats at the bottom of the screen:

```
Date choice:_ 1(month day, year) 2 (dd-mmm-yy) 3 (mm/dd/yy) 4 (dd-mm-yy)
```

Type 2 and then press RETURN to select the second format.

### Create Other Fields

After you have finished creating the *Date* field, move the cursor to create the next field, *Current Balance*, which begins in column 42, line 5. This field must be able to contain up to six digits, with a decimal point (.) as a field marker between the first four digits and the last two.

Since each position in this field is to contain only integers in the range 0 to 9, type 9s, as follows:

```
9999,99
```

Move the cursor down to the next field, *Deposit*, and create that field exactly as you created the *Current Balance* field. Do the same for the *New Balance* field.

The last field you create in DEPOSIT is captioned *Memo*:. To create this field, move the cursor to line 12, column 28. Since this field extends to column 62, it contains 35 characters. Therefore, this field should be defined by 35 consecutive Xs. The X is a field-definition character that tells the Form Driver to permit any displayable character to be entered into the field. You can type in each X individually, or you can use the REPEAT function to enter the 35 Xs more easily.

### Use the REPEAT Function

The REPEAT function simplifies the task of typing in field-definition characters for large fields. To use the REPEAT function, do the following:

1. Press GOLD.
2. Using the numeric keys on your terminal's keyboard, type the number 35. Note that as soon as you type 3, the following prompt appears at the bottom of the screen:

```
Repeat: 3
```

3. Type 5, then X. The field now has 35 Xs.

Now that you have created the fields for DEPOSIT, return to Text mode by pressing TEXT. The Form Editor updates the Modes field in the phase status line.

### Creating Video Attributes for DEPOSIT

You are now ready to assign video characteristics to your form. Two characteristics are used in DEPOSIT:

1. The title, MAKE A DEPOSIT, is in bold.
2. The *Memo* field is underlined.

To make the title bold, do the following:

1. Move the cursor to M in MAKE by pressing either the arrow keys or GOLD TOP, followed by CHARFWD several times.
2. Press SELECT.
3. Move the cursor to the Tin DEPOSIT.
4. Press VIDEO. The Form Editor responds by displaying the VIDEO: prompt at the bottom of the screen.
5. Type BOLD.

VIDEO: BOLD

6. Press ENTER. The Form Editor responds by holding the title and repeating the VIDEO: prompt.
7. Type SAVE to save the video attributes just assigned:

VIDEO: SAVE

8. Press ENTER.

To underline the *Memo* field, do the following:

1. Move the cursor to any character location in the field.
2. Press VIDEO. The Form Editor responds by displaying the VIDEO: prompt at the bottom of the screen.
3. Type UNDERLINE.

VIDEO: UNDERLINE

4. Press ENTER. The Form Editor responds by underlining the *Memo* field and by showing the VIDEO: prompt again.
5. Type SAVE to save the video attribute just assigned:

VIDEO: SAVE

6. Press ENTER.

You are now ready to assign attributes to the fields in DEPOSIT.

Return to the menu by pressing GOLD MENU. The Form Editor clears the screen and displays the menu.

### **3.3.2.3. Assigning Field Attributes to DEPOSIT: The Assign Phase**

You assign attributes to the fields in DEPOSIT while in the Assign phase. Type ASSIGN and then press RETURN. The Form Editor responds by displaying a question at the bottom of the screen (see *Figure 3.5, "First Assign Phase Question"*).

Press RETURN, indicating that you want to assign attributes to the new fields you have created.

The screen is cleared, and DEPOSIT is displayed on the top half of the screen. (The Assign phase questionnaire (see *Figure 3.7, "Display Only Attribute"*) is on the lower half of the screen.)

Note that the *Date* field in DEPOSIT is in reverse video. This helps you identify the field to which you are assigning attributes.

When you create the *Date* field, the Form Editor automatically assigned the Display Only attribute. When DEPOSIT is displayed during run time, the date is displayed and the operator cannot access that field. So the only attribute you need to assign to the *Date* field is the name, DATE. Press LINEFEED to delete the default field name. Type DATE and then press RETURN to assign attributes to the next field.

**Figure 3.7. Display Only Attribute**

```

Assign Field Attributes
Field Name: CURBAL Index Value ____
                                     of ____
- Autotab      - Right Justify  - Uppercase
- No Echo      - X Fixed Decimal  - Must Fill
- Display Only - Zero Fill      - Response Required Clear Character ____
                                     UARs? (Y,N)  N
Default Value:
Help Text:

```

The Form Editor again displays the Assign phase questionnaire. This time the reverse video highlighting is in the *Current Balance* field. The Form Editor has also supplied a default name, F\$0002, for this field,

To enter a name for the *Current Balance* field, first delete the default name by pressing LINEFEED. Then type the name CURBAL.

Next, press TAB to move the cursor to the blank captioned *Display Only*. Type X. The cursor then moves automatically to the next blank, *Right Justify*. Type X. In the fields captioned *Zero Fill* and *Zero Suppress*, type X. The Zero Fill attribute causes the Form Driver to pad with zeros all values returned to the application program. The Zero Suppress attribute causes the Form Driver to provide blanks in place of leading zeros when the current balance figure is displayed during run time.

Finally, type 0 in the blank captioned *Clear Character*. The 0 causes the Form Driver to put Os in each unused data position in the field. When you assign the Zero Suppress attribute, FMS requires you to also use the Zero Fill attribute and to specify a Clear Character. After you have assigned these attributes, press RETURN to move to the next field, *Deposit*.

After typing the name DEPOSIT, press TAB until the cursor is positioned next to the blank captioned *Fixed Decimal*. Type X.

Next, put an X by the *Zero Fill* and *Response Required* attributes. Move the cursor to the blank captioned *Clear Character*. Type 0.

Since the *Deposit* field will be filled in by the operator, you should provide a help text line. To do this, press TAB to move the cursor to the *Help Text* field. Type the following line:

```
Enter amount of deposit
```

Press RETURN to assign attributes to the next field, *New Balance*. Assign the following attributes to this field: the name NEWBAL and the Display Only and Right Justify attributes.

Finally, you assign attributes to the *Memo* field. Assign the name MEMO and the Response Required attribute. You must also type the following help text line:

```
Enter the origin of deposit
```

You have now finished assigning field attributes to the *Deposit* field. Press GOLD MENU to return to the menu.

### 3.3.2.4. Alternative Method of Assigning Field Attributes

You have been assigning field attributes to forms from the Assign phase. The Form Editor offers an alternative to this means of assigning field attributes. After you have created a field and the cursor is still in that field, press GOLD FLDATR. The Form Editor responds by displaying the Assign phase questionnaire on the lower half of the screen. You can immediately assign field attributes. When you complete that questionnaire, you are still in the Layout phase and can continue laying out the form.

You can try this method by returning to the Layout phase for either form you created. Move the cursor to any field on the screen and press GOLD FLDATR. You return to the Form Editor menu by pressing GOLD MENU.

### 3.3.2.5. Testing a Form: The Test Phase

Type TEST and then press RETURN to enter the Test phase. The Form Editor displays DEPOSIT as if it were being displayed by the Sample Application. As you test this form, check the following:

1. Make sure that the visual characteristics are as you intended. Check the position of the title, the background text, and the fields.
2. Press HELP. Check to make sure that the help message appears correctly for each field.
3. Enter invalid data into the fields. Check to make sure that the correct error messages appear.

When you are satisfied that DEPOSIT is correct, press RETURN or ENTER to return to the menu.

If you discover that you need to modify DEPOSIT, reenter any phase – Form, Layout, or Assign – and make the corrections. You will then want to enter the Test phase again.

### 3.3.2.6. Saving a Form: The Exit Phase

After you return to the menu, type EXIT and then press RETURN. The Form Editor responds by displaying the following question:

```
Do you wish to save this form? Y
```

Press RETURN if you wish to save the form you just created. If you type N and then press RETURN, the Form Editor deletes the form you just created.

## 3.4. Creating a Help Form

This section describes how to create a help form. When you ran the Sample Application, you saw the help form associated with the menu (see *Figure 3.8, "Menu Help Form"*).

You will create the help form for MENU and will connect it with MENU.

To create a help form, you follow the same steps as you did to create MENU and DEPOSIT. The only difference between those forms and a help form is that a help form cannot contain any fields, except for time and date fields. Help forms only display text. The Form Driver ignores any other fields that may be in the help form.

The first step is to invoke the Form Editor and to create the form entitled HMENU.

### 3.4.1. Creating HELP\_MENU

Type the following command to invoke the Form Editor and to specify HMENU as the form you wish to create:

```
$ FMS/EDIT HMENU
```

FMS responds by clearing the screen and then displaying the Form Editor menu. Select the Form phase.

**Figure 3.8. Menu Help Form**

```

Help for SAMP Menu

SAMP simulates some functions of electronic banking. In this application
you can make deposits and write checks at your terminal.

Your account at this bank has already been set up. You have made several
deposits and have written some checks. When SAMP starts, you are able to
request the following functions by typing the number and then pressing
RETURN. You can stop SAMP by typing 1 or by pressing keypad period.

1 Exit To leave SAMP and return to operating system level.

2 Write a check You are asked to fill in payee, amount, and memo information.

3 Make a deposit You are asked to fill in the amount and memo information.

4 View check register You can review checks and deposits made in the past.

5 Show account data You can see the data associated with your account.

For more help, press HELP.
To continue, press RETURN.
  
```

When in the Form phase, assign the name HELP\_MENU to this help form. Give this form a white screen background.

Return to the Form Editor menu and select the Layout phase.

When in Layout, type in the text as shown in *Figure 3.8, "Menu Help Form"*. Assign the appropriate video attributes.

When you have finished the Layout phase, return to the menu and exit. You do not need to use the Assign phase unless your help form has a time or a date field. If you were to create fields in a help form, the Form Driver would ignore them during run time.

The next step to take in creating the help form is to associate it with the menu form. To do this, you need to edit MENU with the Form Editor again.



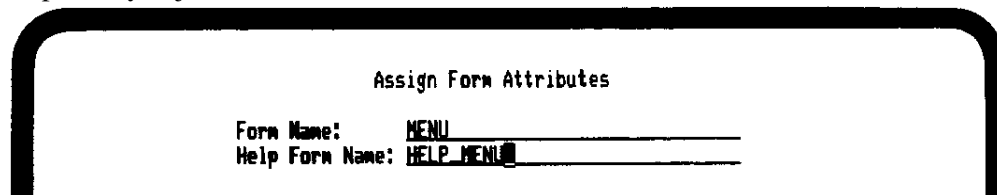
### 3.4.2. Associating `HELP_MENU` with `MENU`

Invoke the Form Editor and specify `MENU` as the form you wish to modify.

```
$ FMS/EDIT MENU
```

FMS responds by clearing the screen and then displaying the Form Editor menu. Select the Form phase.

When in the Form phase, move the cursor to the field captioned *Help Form Name*. Type the name of the help form you just created, `HELP_MENU`.



```

      Assign Form Attributes
Form Name:  MENU
Help Form Name: HELP_MENU

```

`HELP_MENU` is now associated with `MENU`. Press **GOLD MENU** to re turn to the menu and to exit, saving the form.



# Chapter 4. Creating a Form Library

As you read this chapter, you will:

- Create a library file
- Store MENU, DEPOSIT, and HMENU in the library
- Get a directory listing of the library
- Examine a form description of MENU

*Figure 4.1, "FMS Application Development Cycle"* shows the steps in the FMS application development cycle. The shaded portions are covered in this chapter.

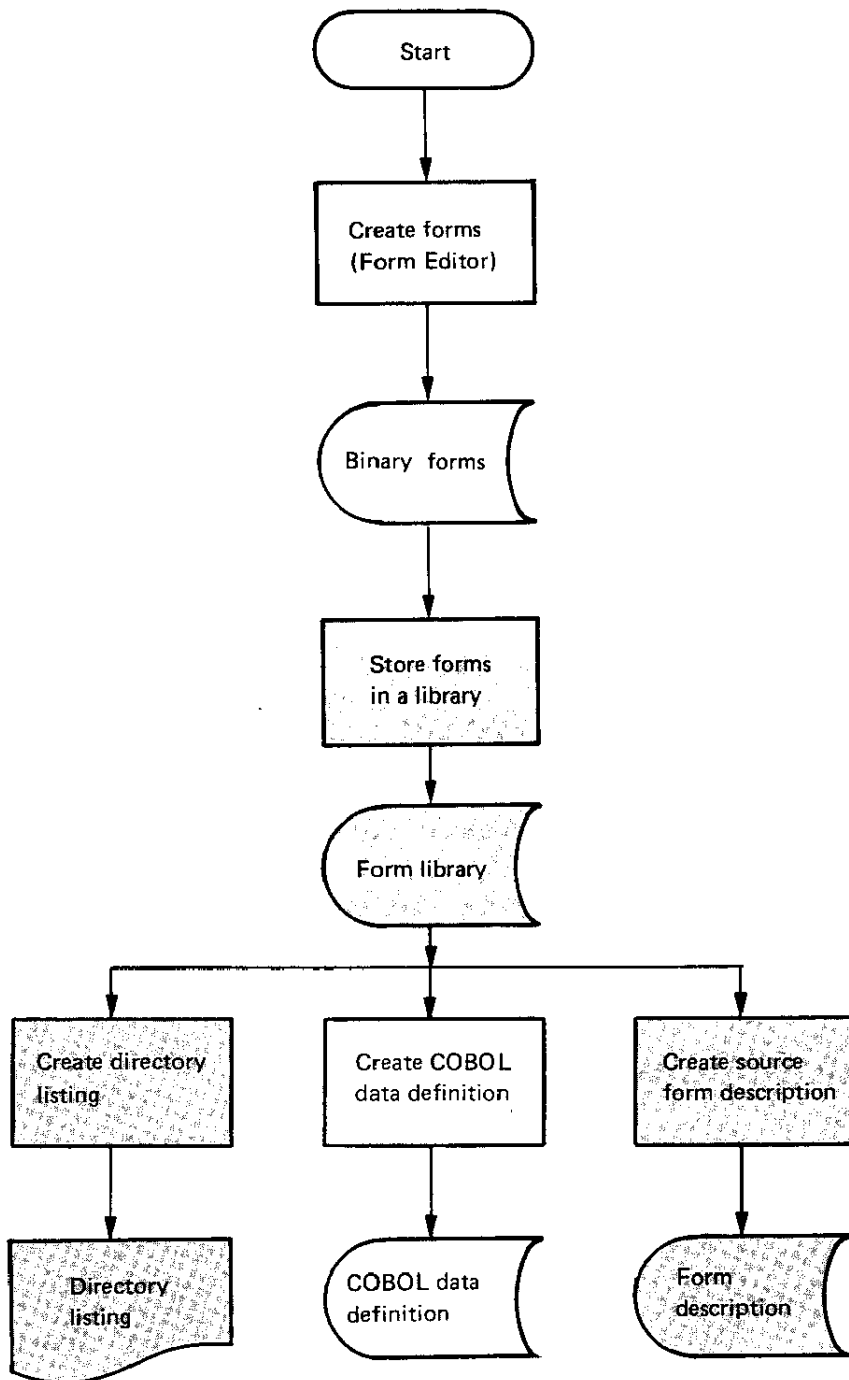
## 4.1. Create a Library File

To create a library for the forms you just created, invoke the Form Librarian and specify SUBSET.FLB as the form library file you wish to create:

```
$ FMS/LIBRARY/CREATE SUBSET,FLB
```

FMS responds by printing the \_Files: prompt. Enter the following command:

```
_Files: MENU,DEPOSIT,HMENU
```

**Figure 4.1. FMS Application Development Cycle**

## 4.2. Obtain a Library Directory Listing

Once you have stored the three forms in SUBSET.FLB, get a directory listing of that library file by specifying /DIRECTORY in the FMS command line. Use the following command line to obtain a directory listing of SUBSET.FLB.

```
$ FMS/DIRECTORY SUBSET
```

FMS then displays the following directory listing:

```
Form Librarian V2.0
```

21-SEP-1882 10:06

Library USER:[SMITHJSUBSET,FLB;1, created: 21-SEP-1882 10:06

Date and time of last modification: 21-SEP-1882 10:08

MENU  
DEPDSIT  
HELP-MENU

## 4.3. Interpret a Form Description

To obtain a form description of MENU.FRM, type the following command:

```
$ FMS/DESCRIPTION SUBSET/FORM_NAME=MENU/OUTPUT=MENU
```

The command above produces the form description MENU.FLG, shown in this section. For easier reference, we will discuss the form description in sections. Form descriptions are useful as hard-copy references of forms. For more information on form descriptions, see the *VSI FMS Utilities Reference Manual*

A form description is a set of Form Language statements that describe a form. The following section of MENU.FLG consists of statements that describe the form attributes of **MENU**.

```
!           FMS Form Description Application Aid
!           Version 2.0

FORM NAME='MENU'
HELP-FORM='HMENU'
  AREA_TO_CLEAR=1:23
  WIDTH=80
  BACKGROUND=WHITE
  DBLWID=7
  DBLSIZ=2
;
```

The next section of MENU.FLG consists of statements that describe the background text in MENU – the location and video attributes of each bit of background text.

```
TEXT (2,9) 'Check ins Account Menu'
  BOLD
TEXT (7,1) '          Choose option C 1-5):
;
TEXT (8,27) '1 Exit'
;
TEXT (11,27) '2 Write a check'
;
TEXT (13,27) '3 Make a deposit'
;
TEXT (15,27) '4 View the check register'
;
TEXT (17,27) '5 Show account data'
;
TEXT (20,48) 'lqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqk'
  CHARACTER_SET=RULE
;
```

```
TEXT (21,48) 'x'
    CHARACTER_SET=RULE
;
TEXT (21,50) 'For help, Press HELP.'
;
TEXT (21,80) 'x'
    CHARACTER_SET=RULE
;
TEXT (22,49) 'x'
    CHARACTER_SET=RULE
;
TEXT (22,50) 'To continue, Press Keypad 1-5.'
;
TEXT (22,80) 'x'
    CHARACTER_SET=RULE
;
TEXT (23,49) 'mqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqq'
    CHARACTER_SET=RULE
;
```

The last section of the form description describes the field in MENU – the position of the field on the screen and the field attributes.

```
ATTRIBUTE-DEFAULTS FIELD
    CLEAR-CHARACTER=' '
    NOAUTOTAB BLANK-FILL NOBLINKING NOBOLD NOREVERSE
    NOUNDERLINE NODISPLAY_ONLY ECHO NDFIXED_DECIMAL
    LEFT_JUSTIFIED NOSUPERVISOR_ONLY NOSUPPRESS NOUPPERCASE
;

FIELD NAME='OPTION' (7,1)
    PICTURE='S'
    HELP='Enter one of the numbers 1, 2, 3, 4, or 5'
    DEFAULT='2' RESPONSE_REQUIRED UNDERLINE

END_OF_FORM NAME='MENU';
```

## 4.4. Obtain a Form Image

The last section of this chapter tells how to obtain a form image and how to interpret it.

To obtain an image of the form MENU, enter the following command:

```
$ FMS/DESCRIPTION/IMAGE SUBSET/FORM_NAME=MENU/OUTPUT=MENU
```

FMS produces the file MENU.LIS shown below.

```

Form: MENU

      1      2      3      4      5      6      7      8
123456789012345678901234567890123456789012345678901234567890
-----
11|                                     |11
21|                                     |12
31|          CHECKING ACCOUNT MENU   |13
41|                                     |14
51|                                     |15
61|                                     |16
71|          Choose option (1-5):      |17
81|                                     |18
91|          1  Exit                    |19
101|                                     |20
111|          2  Write a check           |21
121|                                     |22
131|          3  Make a deposit         |23
141|                                     |24
151|          4  View the check register|25
161|                                     |26
171|          5  Show account data      |27
181|                                     |28
191|                                     |29
201|          +-----+                |30
211|          |For help, press HELP (PF2).|31
221|          |To continue, type a number (1-5) and press RETURN.|32
231|          +-----+                |33
-----
123456789012345678901234567890123456789012345678901234567890
      1      2      3      4      5      6      7      8

```

This image shows how MENU appears on the screen. The title, CHECKING ACCOUNT MENU, and the line Choose Option are not shown in double-size and double-wide characters, because of the limitations of a lineprinter. The image shows the location of background text and fields that are in normal-size characters. When working with hard-copy listings of forms, it is useful to attach copies of the form image to the form descriptions.





# Chapter 5. Writing an Application

As you read this chapter, you will:

- Learn about some basic Form Driver concepts
- Write a subset of the Sample Application, using the three forms you created in *Chapter 3, "Creating Forms"*
- Learn about some basic Form Driver calls and incorporate them in the subset application

## 5.1. Form Driver Concepts

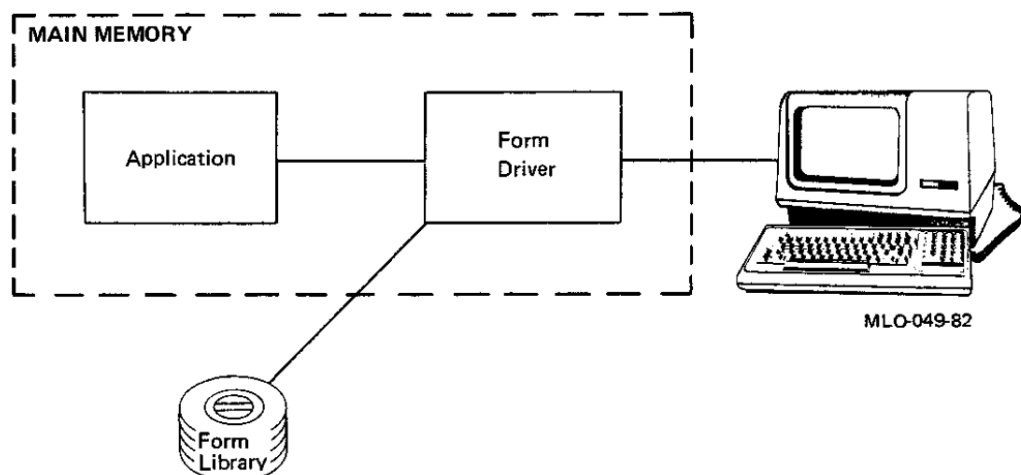
The Form Driver consists of subroutines that provide services to make it easy for an application to display forms and to transfer data between the terminal and the Form Driver. The Form Driver services are invoked by Form Driver calls in the application. The Form Driver routines accomplish the following tasks:

- Establish communication links among the Form Driver, the application, the terminal, and the form library
- Display the forms on the screen
- Accept operator input into the fields of the form
- Determine whether the operator input is valid
- Send the operator input back to the application
- Move the cursor from one field to another
- Display messages and help for the operator

You implement Form Driver services by including Form Driver calls in the source program. You then compile and link the application program with the Form Driver routines. During run time, the Form Driver routines are invoked in much the same way as routines available from any object program library.

As shown in *Figure 5.1, "Role of the Form Driver in an Application Program"*, the Form Driver is a part of the application program. Form Driver calls set up all the I/O channels and work areas that you specify.

**Figure 5.1. Role of the Form Driver in an Application Program**



In this section, you will learn about the categories of Form Driver calls. In the programming exercises in this chapter, you will use the calls that are demonstrated in the Sample Application. Finally, you will learn how the Form Driver performs string handling. Understanding string handling is important when adapting your programming language to FMS.

## 5.1.1. Form Driver Calls

Form Driver calls are implemented in the application program in much the same way as any other subroutine calls. The format of the calls depends on the program language used. This section briefly describes the kinds of calls and lists the calls used in the subset application.

### 5.1.1.1. Functional Division of Calls

Because the Form Driver provides a number of services, the calls can be divided into the following categories:

- Control calls
- Field-level calls
- Form-level calls
- Utility calls

The services that the calls provide are listed under each category.

#### Control Calls

- Establish communication lines with the terminal and the form library
- Establish work areas for forms and the terminal
- Manipulate files and work areas
- Detach communication paths previously established

#### Field-Level Calls

- Request operator input to a field
- Alter characteristics of a field
- Send data to a field
- Return a description of a field
- Return a field's context, name, and other data

#### Form-Level Calls

- Clear the terminal screen
- Display a form

- Return all field values in one step
- Send values to all fields in one step

#### Utility Calls

- Cancel a call
- Send a message to the terminal
- Return a line from the terminal
- Return Named Data from a form
- Return status after the last call executed

### 5.1.1.2. Form Driver Calls Used in the Subset Application

In this chapter, you will practice using Form Driver calls. After completing this chapter, you can turn to the *VSI FMS Form Driver Reference Manual* to write your own FMS applications.

In the call descriptions in this chapter, some arguments for some of the calls have been omitted for the sake of simplicity. The *VSI FMS Form Driver Reference Manual* provides complete descriptions of each call.

The following calls are used in the exercises:

FDV\$ATERM	Attach terminal
FDV\$AWKSP	Attach form workspace
FDV\$CDISP	Clear screen and display form
FDV\$DTERM	Detach terminal
FDV\$GET	Get value for specified field
FDV\$LCLOS	Close form library
FDV\$LOPEN	Open form library
FDV\$PUT	Output value to specified field
FDV\$PUTL	Output line to screen
FDV\$STAT	Return status from the last call
FDV\$WAIT	Wait for operator

### 5.1.2. String Handling

This section describes how the Form Driver manipulates data that it gets from the operator. Your application accesses data from forms in a variety of ways and can do the following:

- Use the Form Driver FDV\$GET call to request operator input to a specific field and return that data in a variable to the application and the Form Driver workspace.
- Use the Form Driver FDV\$GETAL call to request input from all the fields in which the operator can provide input – readable fields – on the form and put that data in the Form Driver workspace. Your application can then access the workspace to return individual field values.

- Get operator input to all readable fields in a form and store the contents of all fields, both readable and nonreadable, as a single string in the application's data base. This is done with the FDV\$GETAL call, specifying the variables to which all field values are to be returned.

This chapter shows how to implement some of the Form Driver's string-handling capabilities that appear in the Sample Application. How an application handles strings depends on the language used, however. For specifics on how to handle strings in a given language, see the *VSI FMS Language Interface Manual*.

## 5.2. Sample Application Subset

This section describes the main program and subroutines of the subset application that you will code in this chapter. The subset is made up of the following program sections:

- A main program
- A subroutine that displays and services the MENU form
- A subroutine that executes the DEPOSIT form

You will also insert dummy subroutines, called stubs, whose primary purpose is to appear where they do in the Sample Application but without any particular function. You will provide stubs where the subset application performs the following tasks:

- Initialize account data
- Process the check-writing form
- Process the deposit form
- Show the account data
- Show the check register

Instead of performing the tasks shown above, the stubs, when called during run time, will display a message on the screen, indicating that their function is not implemented yet. Providing these stubs in the subset application enables you to run the subset and to test all the options available in the menu.

---

### Note

To make efficient use of VAX-11 BASIC, each subroutine in the subset application is invoked as a function, as in the Sample Application.

---

The subset application does not use the keypad. Therefore, when you run the subset application, you will need to indicate menu choices by using the numeric keyboard.

### 5.2.1. The Main Program

The main program of the subset application does the following:

1. Sets up a general work area for the program.
2. Sets up a terminal control area for the Form Driver.

3. Attaches the terminal to the Form Driver.
4. Attaches the workspace to the terminal.
5. Establishes a communication line between the form library and the Form Driver.
6. Calls a subroutine, INACCT, that reads account data from a data file. Since this subroutine will not be coded in this chapter, it will return to the main program only arbitrary account figures.
7. Calls a subroutine, MENU, that processes all menu requests.
8. Performs a general cleanup.
9. Exits.

## 5.2.2. Subset Subroutines

Following are descriptions of the subroutines that you will include in the subset from the Sample Application.

**INACCT** Returns account data to the main program. In this subset, INACCT returns only the summary figures for the checking account: starting balance, total checks written, total deposits made, and current balance.

**MENU** Accepts operator input from the menu form and branches control to the appropriate subroutine to service the action requested. Control always returns to this subroutine until the operator exits. Then this subroutine returns control to the main program. This subroutine uses the form MENU that you created in *Chapter 3, "Creating Forms"*.

**WRITCH** Performs the check-writing procedure in the Sample Application. In the subset, this subroutine is a stub and displays a message indicating that the subroutine is not implemented. The WRITCH subroutine then returns control to the MENU subroutine.

**MAKDEP** Instructs the operator to enter a deposit and a memo of the deposit. This is the second subroutine for which you created a form in *Chapter 3, "Creating Forms"*. This subroutine makes use of the keypad; if the operator presses keypad period, the Form Driver displays the MENU form and returns control to subroutine MENU.

**VUEREg** Permits the operator to view the check register for the checking account. In the subset, this subroutine displays a message and returns control to subroutine MENU.

**VUEACT** Permits the operator to view the account data. This subroutine is not implemented in this subset. Therefore, when the operator chooses this option, VUEACT displays a message that this subroutine is not implemented.

**GETSTA** Checks the status of the Form Driver after the last call executed. This subroutine displays an error message if the results of the last Form Driver call executed produced an error. The final statement of the application is at the end of this subroutine.

## 5.2.3. Preparing the Main Program

Before you continue with this section, create a file in which you can enter the statements given in this chapter. Use the name SUBSET.BAS for this file. As you are instructed to enter statements, enter them as shown in SUBSET.BAS.

### 5.2.3.1. Initializing Calls

The first instructions in the subset's main program are the initializing calls. These calls set up the Form Driver, the form library, the workspace, the terminal control area, and the appropriate communication links among those components.

You use the initial Form Driver calls to do the following:

- Attach the terminal to the Form Driver
- Attach a workspace for the Form Driver
- Open the form library containing the forms the application will use

In this section, you will write the statements that use these calls and see how the calls work.

Descriptions of the calls used in the subset are indented. The statements you enter into the source file for SUBSET.BAS appear in red type. Note that the complete subset program appears in the *Appendix A, "Subset Application Listing"*.

#### Attaching the Terminal – FDV\$ATERM

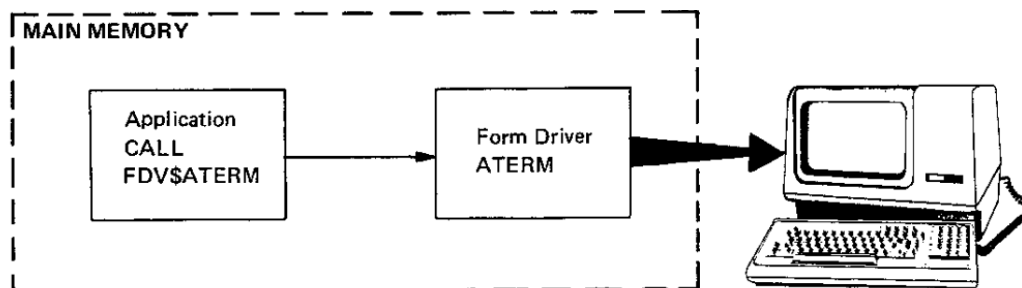
Before an FMS application and a terminal can communicate, the terminal must be identified. A terminal control area, an area in memory used by the Form Driver, must also be established for that terminal.

The FDV\$ATERM call identifies the terminal and establishes a terminal control area (see *Figure 5.2, "Attaching a Terminal"*). The FDV\$ATERM call has the following format:

#### FDV\$ATERM (tea, size, channel)

The argument **tea** represents an area in memory set aside by the application program for the terminal control area, **size** represents the size, in bytes, of the TCA, and **channel** represents the I/O channel.

**Figure 5.2. Attaching a Terminal**



Enter the following statements into SUBSET.BAS to set up a general work area and a terminal workspace and to attach the terminal:

```

130      DIM WORKSPACE% ( 3 )      ! General workspace
140      DIM TCA% ( 3 )            ! Terminal control area 12%, 2% I
1040     CALL FDV$ATERM ( TCA% ( ), 12%, 2%)
  
```

The value **3** in the DIM statements for WORKSPACE% and TCA% represents a 12-byte area in memory. VAX-11 BASIC interprets the value 3 to represent three 4-byte words, or 12 bytes. In the FDV \$ATERM call, the argument **12%** represents the size, in bytes, of the TCA, and **2%** represents the I/O channel number.

### Attaching the Workspace – FDV\$AWKSP

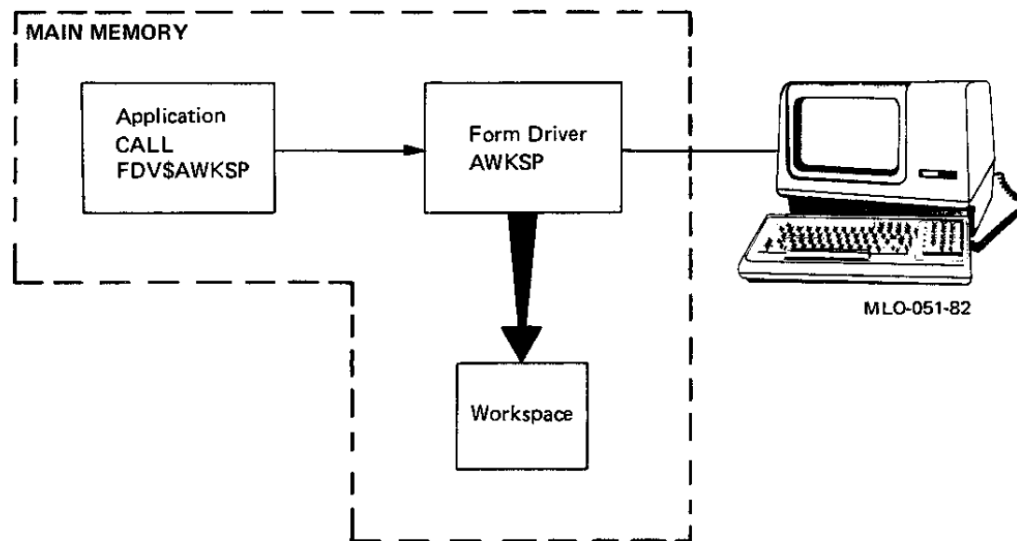
To display a form on a terminal screen, your program must put a form description in a workspace and call the form from that workspace.

The Form Driver call that attaches a workspace to a terminal is FDV\$AWKSP (see *Figure 5.3, "Attaching a Workspace"*). The FDV\$AWKSP call has the following format:

**FDV\$AWKSP (wksp, size)**

The argument **wksp** represents the area in memory set aside for the workspace, and **size** represents the size, in bytes, of the workspace.

**Figure 5.3. Attaching a Workspace**



Enter the following statement to attach a workspace size of 2000 bytes to the terminal:

```
1042 CALL FDV$AWKSP ( WORKSPACE% (), 2000% ) \ C=FN,GETSTA
```

The accuracy of the initial estimate of the workspace size is not critical, but applications run slightly faster when the size is larger than the workspace requirements of the largest form in the application. To determine the size, in bytes, of each form in a library, use the FMS/LIBRARY/FULL command.

In the statement above, C=FN.GETSTA calls the FMS status-checking function, which is described later. It is good programming practice to check the FMS status after Form Driver calls that can affect subsequent processing – attaching, opening, or displaying operations.

### Opening a Library Channel – FDV\$LOPEN

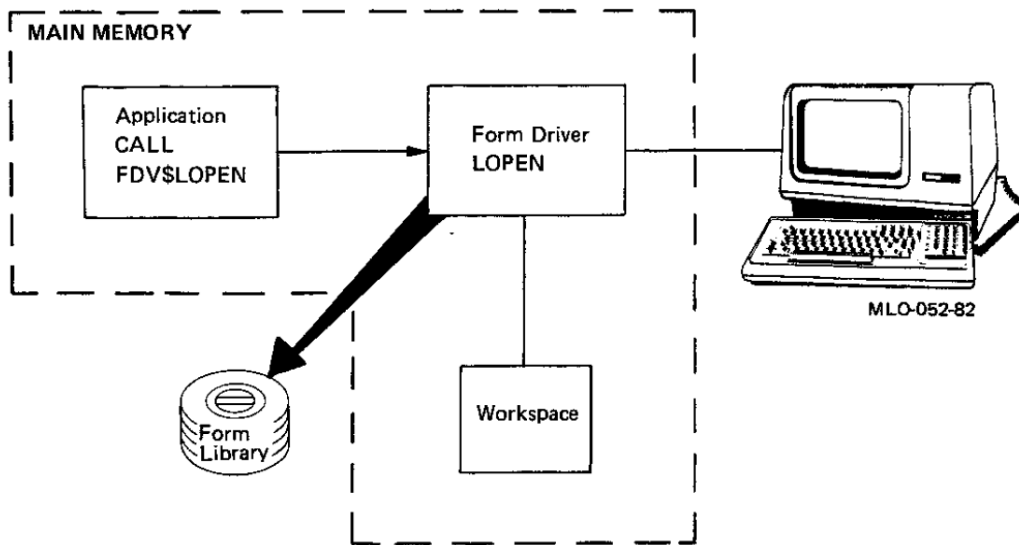
Forms are used to transfer data between the application program and the terminal screen. Forms may be memory resident – linked with the application program – or stored in a form library. To retrieve a form stored in a library, the library must be opened, and an I/O channel connecting the library and the Form Driver must be established.

The FDV\$LOPEN call establishes an I/O channel between the Form Driver and the form library (see *Figure 5.4, "Opening a Library Channel"*). The FDV\$LOPEN call has the following format:

**FDV\$LOPEN ( filspc, channel )**

The argument **filspc** represents the name of the library file that you wish to open, and **channel** represents the I/O channel that connects the library file to the Form Driver.

**Figure 5.4. Opening a Library Channel**



Enter the following statement to open the library SUBSET.FLB and to specify an I/O channel to be used between the library and the Form Driver:

```
1050 CALL FDV$LOPEN ( 'SUBSET', 1% ) \ C=FN,GETSTA
```

The GETSTA function call appears again at the end of the statement.

### 5.2.3.2. Coding the Body of the Main Module

You will now enter the statements for coding the body of the subset main module. In the subset, the body consists of two statements: a call to the subroutine INACCT and a call to the subroutine MENU. INACCT initializes the account information. In this subset, INACCT establishes a current balance for the checking account and returns control to the main module. MENU processes all menu requests.

Enter the following statements:

```
1115 C=FN.INACCT      ! Initialize account information
1170 C=FN.MENU        ! Process all menu requests
```

When the operator exits from the menu, control returns to the main program, and the closeout procedure statements are processed.

### 5.2.3.3. Closing Calls

At the end of the main module, several statements execute the closing procedure for the FMS components. These statements close the I/O links opened in the main module, detach the terminal and workspace(s), and perform a general cleanup.

The closing procedure calls are as follows:

FDV\$LCLOS	Close form library
FDV\$DWKSP	Detach form workspace



FDV\$DTERM	Detach terminal
------------	-----------------

### Closing the Form Library – FDV\$LCLOS

The FDV\$LCLOS call closes the I/O channel established between the library file and the Form Driver. FDV\$LCLOS has the following format:

#### FDV\$LCLOS

This call has no arguments.

### Detaching the Form Workspace – FDV\$DWKSP

The FDV\$DWKSP call detaches the workspace from the application program. FDV\$DWKSP has the following format:

#### FDV\$DWKSP (wksp)

The argument **wksp** represents the area of memory set aside for the Form Driver workspace.

### Detaching the Terminal – FDV\$DTERM

The FDV\$DTERM call detaches the operator's terminal from the application program. This call also cleans up the terminal and makes it usable for other programs. FDV\$DTERM has the following format:

#### FDV\$DTERM (tea)

The argument **tea** represents the area of memory set aside for the terminal control area.

Enter the following statements to execute the FMS closing procedure:

```
1200 CALL FDV$LCLOS
1208 CALL FDV$DWKSP ( WORKSPACE% () )
1215 CALL FDV$DTERM ( TCAZ () )
1220 GOTO 15999
```

The last line, GOTO 15999, transfers control to the end statement of the subset.

## 5.2.4. Coding the INACCT Subroutine

The INACCT subroutine returns to the main program the following arbitrary figures:

SBALANCE%	The starting balance: \$503.75
TOTPAY%	The total amount written in checks: \$213.45
TOTDEP%	The total amount of deposits made: \$1123.23
BALANCE%	The current balance: \$1413.53

Each variable is initially assigned a numeric value. Later in this chapter, you will see how these numbers are converted into strings that are output to the appropriate forms in the subset application. The INACCT subroutine has no Form Driver calls.

Enter the following statements, which make up the body of the INACCT subroutine.

```
4000 DEF FN.INACCT
```

```

4005  SBALANCE% = 50375
4010  TOTPAY% = 21345
4015  TDTDEP% = 112323
4020  BALANCE%= 141353
4080  FNEND

```

## 5.2.5. Coding the MENU Subroutine

The MENU subroutine does the following:

1. Accepts input from the menu form and branches control to the appropriate routine
2. Continues accepting input until the operator enters 1 (Exit)

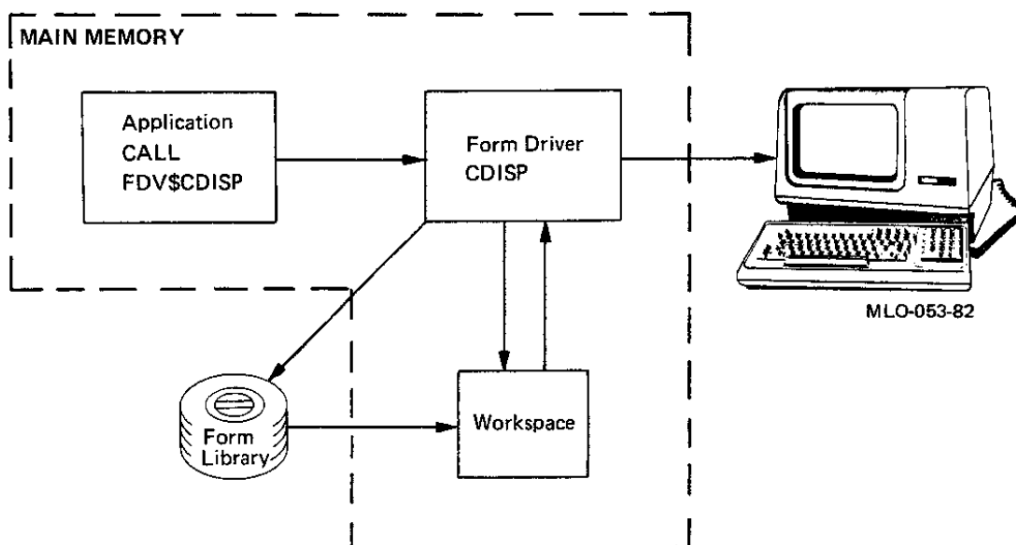
Before entering the code for this subroutine, become familiar with the Form Driver calls used in this subroutine.

### Clearing the Screen and Displaying a Form – FDV\$CDISP

The FDV\$CDISP call clears the terminal screen and displays a specified form. In the MENU subroutine, FDV\$CDISP displays the menu form that you created in *Chapter 3, "Creating Forms"*.

When processing the FDV\$CDISP call, the Form Driver retrieves the specified form from the library, places a description of that form in the workspace, clears the terminal screen, and displays the form on the terminal screen (see *Figure 5.5, "Displaying a Form"*).

**Figure 5.5. Displaying a Form**



The FDV\$CDISP call has the following format:

**FDV\$CDISP ( frmnam )**

The argument **frmnam** represents the name of the form that the Form Driver is to display on the screen.

In the subset, the corresponding statement for this call is as follows:

**CALL FDV\$CDISP ( 'MENU' )**

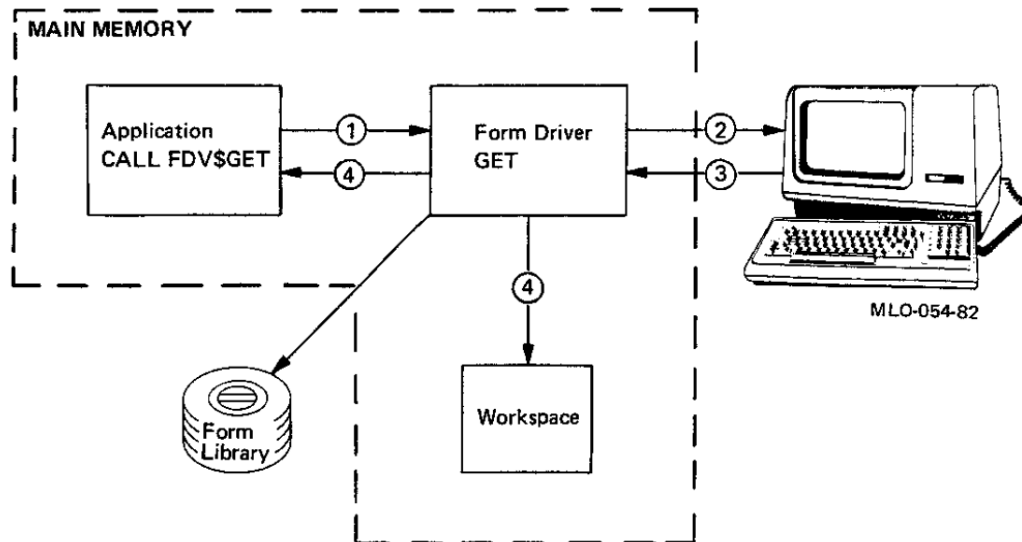
You will be instructed later to enter this statement.

### Requesting Operator Input to a Field – FDV\$GET

The FDV\$GET call requests the operator's input to a specified field. In the MENU subroutine, the GET call is used to request the operator's choice of checking account procedure to enter.

When processing the FDV\$GET call, the Form Driver reads the operator's input to a specified field, copies that input to a variable name specified in the GET call statement, and sends that variable to the application (see *Figure 5.6, "Displaying a Form"*).

**Figure 5.6. Displaying a Form**



The Form Driver uses the following procedure when executing the FDV\$GET call:

1. The application issues the FDV\$GET call for a value.
2. The Form Driver requests the value from the operator.
3. The Form Driver retrieves the value.
4. The Form Driver copies the value into the workspace and into the application program.

The FDV\$GET call has the following format:

**FDV\$GET ( fldval, fldtrm, fldnam )**

The argument **fldval** represents the variable into which the input to the specific field is copied, **fldtrm** represents the code for the field terminator key struck, and **fldnam** represents the name of the field from which the input is being retrieved.

The subset application call that requests operator input to the menu has the following format:

**CALL FDV\$GET ( OPTION\$, TERMINATOR%, 'OPTION'**)

### Displaying a Line on the Terminal Screen – FDV\$PUTL

The FDV\$PUTL call displays a message on a specified line on the terminal screen. In the MENU subroutine, the FDV\$PUTL call is used to display a message indicating invalid input by the operator into the OPTION field.

The FDV\$PUTL call has the following format:

**FDV\$PUTL ( text [,line] )**

The argument **text** represents the character string to be displayed on the screen, and **line** represents the number of the line on which the character string is to be displayed. If **line** is not supplied, the Form Driver displays the character string on the bottom line of the screen.

If the operator attempts to enter an invalid number, the message "INVALID CHOICE" is displayed by the following call:

**CALL FDV\$PUTL ( 'INVALID CHOICE' )**

### **Coding the Body of the MENU Subroutine**

The MENU subroutine uses a loop to continue displaying the menu after processing an operator choice. A computed GOTO dispatches to the subroutine which processes the operator's request.

Enter the following statements, which make up the body of the **MENU** subroutine.

```
5000 DEF FN.MENU
5005 !      Menu choices:
5010 ! 1 => Exit
5015 ! 2 => Write checks
5020 ! 3 => Make a deposit
5025 ! 4 => View register
5030 ! 5 => View account data
```

As required in VAX-11 BASIC, the following statement pre-extends the variable OPTION\$, which is to contain the operator menu choice:

```
85040 OPTION$ = ' '
```

The following statements make up a computed GOTO loop to dispatch to the appropriate subroutine. Calls to the status check function appear after the FDV\$CDISP and FDV\$GET calls.

```
5045 WHILE 1 = 1
5050     CALL FDV$CDISP ( 'MENU' ) \ C=FN.GETSTA
5070     CALL FDV$GET ( OPTION$, TERMINATOR%, 'OPTION' ) \ C=FN.GETSTA
5075     ON VAL ( OPTION$ ) GOTO 5082, 5090, 5100, 5110, 5120
```

The remaining statements of MENU branch control to the subroutine(s) processing the operator's menu request. After a subroutine has been executed, control resumes at statement 5130, which branches back to the beginning of the computed GOTO loop.

```
5081     ! Option 1: Exit
5082     FNEXIT
5085     ! Option 2: Write checks
5090     C=FN.WRITCH \ GOTO 5130
5095     ! Option 3: Make a deposit
5100     C=FN.MAKDEP \ GOTO 5130
5105     ! Option 4: View register
5110     C=FN.VUEREG \ GOTO 5130
5115     ! Option 5: View account data
5120     C=FN.VUEACT \ GOTO 5130
5130 NEXT
5140 FNENO
```

## 5.2.6. Coding the WRITCH Subroutine

The WRITCH subroutine does the following:

1. Displays the following message on the bottom of the screen:

```
Write a check - not implemented yet
```

2. Waits for the operator to press ENTER or RETURN
3. Returns to the main menu

WRITCH uses two Form Driver calls: FDV\$PUTL and FDV\$WAIT. Here we will discuss only the FDV\$WAIT call.

### Waiting for Operator Input – FDV\$WAIT

The FDV\$WAIT call causes the Form Driver to wait until the operator signals to proceed. This call is particularly useful for pausing until the operator reads a message. The FDV\$WAIT call has the following format:

#### FDV\$WAIT ( [fldtrm] )

The argument **fldtrm** is the variable to which the Form Driver returns the terminator code of the key struck to end the wait condition. If you do not supply an argument with the WAIT call, the Form Driver will not return the terminator the operator entered.

### Coding the Body of WRITCH

The WRITCH subroutine uses the FDV\$PUTL call to display the following message on the bottom of the screen:

```
Write a check - not implemented yet
```

Enter the following statements to make up the WRITCH subroutine:

```
11000 DEF FN,WRITCH
11005 CALL FDV$PUTL ( 'Write a check - not implemented Yet' )
11010 CALL FDV$WAIT
11015 FNEND
```

## 5.2.7. Coding the MAKDEP Subroutine

The MAKDEP subroutine does the following:

1. Displays the DEPOSIT form, which you created in *Chapter 3, "Creating Forms"*
2. Writes the checking account's current balance in the *Current Balance* field
3. Requests the operator's input for the deposit and the deposit memo
4. Adds the deposit to the current balance and updates the *Current Balance* field
5. Displays, once the deposit has been entered, the following message on the bottom of the screen:

```
Deposit Made, Press RETURN or ENTER to continue
```

The MAKDEP subroutine uses the following Form Driver calls, which we have already presented:

FDV\$CDISP	Clears the screen and displays the DEPOSIT form
FDV\$GET	Gets the operator's input for the <i>Enter Deposit</i> and <i>Memo</i> fields
FDV\$PUTL	Displays a message on the bottom line of the screen, saying that the deposit has been made
FDV\$WAIT	Waits for the operator to signal for the Form Driver to continue after the deposit message has been displayed

The MAKDEP subroutine also uses the FDV\$PUT call to display the current balance, determined previously by the INACCT subroutine or by previous calls to MAKDEP, in the *Current Balance* field.

### Output Value to a Specified Field – FDV\$PUT

The FDV\$PUT call displays a value in a specified field. In the MAKDEP subroutine, the FDV\$PUT call is used to display the date and the current balance in the *Date* and *Current Balance* fields, respectively.

The FDV\$PUT call has the following format:

**CALL FDV\$PUT ( fldval, fldnam [,fldidx] )**

The argument **fldval** represents the variable to be displayed, **fldnam** represents the field in which the variable is to be displayed, and **fldidx** represents the index value of the field represented by **fldnam**. (The argument **fldidx** is used only if the field is indexed, described in *Chapter 7, "Programming Features"*.)

The statement that displays the current balance in the subset application follows:

```
CALL FDV$PUT ( STR$ C BALANCE %, 'CURBAL' )
```

(Note that the BASIC STR\$ function is used in this statement, described later.)

### BASIC Functions in the MAKDEP Subroutine

The BASIC functions STR\$, SPACE\$, and VAL are described here for those who are not familiar with VAX-11 BASIC. These functions are used in MAKDEP to perform the following operations:

STR\$	Converts a numeric value to a string value. For example, the current balance, represented by the variable CURBAL, initially has the numeric value 50375. The STR.\$ function converts the number 50375 to the string '50375'. The Form Driver deals only with character strings, not with numbers, for values of fields.
SPACE\$	Creates a string of space characters, as many as specified.
VAL	Extracts the numeric value from a string. For example, the VAL function can extract the value 50375 from the string '50375' for subsequent arithmetic operations. This function does the reverse of the STR\$ function.

### Coding the Body of MAKDEP

Statements 12000 to 12065 display the form DEPOSIT and send the current balance to that form. Calls to the FMS status-checking function appear at the end of these statements.

```
12000 DEF FN.MAKDEP
12050 CALL FDV$CDISP ( 'DEPOSIT' ) \ C=FN.GETSTA
12065 CALL FDV$PUT ( STR$C BALANCE% ), 'CURBAL' ) \ C=FN,GETSTA
```

Statements 12125 and 12130 define two variables: DEP.AMT\$, which represents the amount of the deposit made; and DEP.MEMO\$, which represents the deposit memo. In these statements, the SPACE\$ function is used to pre-extend DEP.AMT\$ and DEP.MEMO\$. These string variables must be at least as long as the return values expected. Thus, BASIC string variables must be pre-extended to that length before being passed by the Form Driver. The SPACE\$ function pre-extends strings by giving them a specific number of spaces.

```
12125 DEP.AMT$ = SPACE$(6)
12130 DEP.MEMO$ = SPACE$(35)
```

Statements 12135 and 12140 use the FDV\$GET call to request the operator to enter data into the fields DEPOSIT and MEMO and then to return those values to variables DEP.AMT\$ and DEP.MEMO\$.

```
12135 CALL FDV$GET ( DEP.AMT$, TERMINATOR%, 'DEPOSIT' )
12140 CALL FDV$GET ( DEP.MEMO$, TERMINATOR%, 'MEMO' )
```

Statement 12155 updates the checking account's current balance, represented by the variable BALANCE%.

```
12155 BALANCE%= BALANCE%+ VAL ( DEP.AMT$ )
```

Statement 12160 uses the FDV\$PUT call to display the updated current balance in the field NEWBAL and then to call the FMS status checking function.

```
12160 CALL FDV$PUT( STR$( BALANCE% ), 'NEWBAL' ) \ C=FN,GETSTA
```

Statement 12165 uses the FDV\$PUTL call to display a message that the deposit has been made.

```
12185 CALL FDV$PUTL ( 'Deposit made - Press RETURN or ENTER to continue' )
```

Statement 12170 uses the FDV\$WAIT call to pause and wait for the operator to press RETURN to continue. Statement 12175 ends the subroutine.

```
12170 CALL FDV$WAIT
12175 FNENO
```

## 5.2.8. Coding Subroutines VUEREG and VUEACT

These two subroutines are stubs in the subset. They display the message that the subroutines are not implemented and then return to the menu, as in the functions of subroutine WRITCH. The statements for both subroutines follow:

```
13000 DEF FN.VUEREG
13005 CALL FDV$PUTL ( 'View register not implemented yet' )
13010 CALL FDV$WAIT
13015 FNEND
14000 DEF FN,VUEACT
14005 CALL FDV$PUTL ( 'View account not implemented yet' )
```

```
14010  FDV$WAIT
14015  FNEND
```

## 5.2.9. Coding Subroutine GETSTA

GETSTA, a subroutine called frequently throughout the subset, checks the current FMS status and diagnoses any errors. Providing a status-checking subroutine in your FMS applications is good programming practice. If an error condition exists, this subroutine prints a code for the specific error condition. (See the *VSI FMS Form Driver Reference Manual* for a list of the codes.)

The GETSTA subroutine does the following:

1. Issues the FDV\$STAT call.
2. If the status code is greater than 0, the last call was successful, so control returns to the statement following the call to GETSTA. If the status code is equal to or less than 0, the FDV\$DTERM call is executed, and codes are printed for both the FMS and the RMS (Record Management System) status. RMS returns any file or system operation error codes to the Form Driver. For more details on RMS, see the *VSI FMS Form Driver Reference Manual*.
3. Terminates the subset application. The END statement, number 15999, is the last statement in the program. Control always passes to this statement. If the previous Form Driver call does not yield an error, control always branches back to the statement following the GETSTA function call before the END statement is reached.

The GETSTA subroutine uses two Form Driver calls: FDV\$STAT and FDV\$DTERM.

### Returning the Status from the Last Call – FDV\$STAT

The FDV\$STAT call returns the status for the last Form Driver call executed. The FDV\$STAT call has the following format:

**FDV\$STAT ( status [,iostat])**

The argument **status** represents a numeric code for the last Form Driver call executed, and **iostat** represents a numeric RMS status code for detailed information when the STATUS value is -4 or -18.

### Detaching the Terminal – FDV\$DTERM

At the end of an application, the FDV\$DTERM call is used to detach the operator's terminal from the application program. The call also cleans up the terminal for use by other programs. The FDV\$DTERM call has the following format:

**FDV\$DTERM ( tea )**

The argument **tea** represents the area of memory set aside for the Form Driver workspace.

### Coding the Body of GETSTA

Statement 15025 uses the FDV\$STAT call to return the FMS and RMS status codes after the last call executed. The FMS status code is represented by FMSSTATUS%, and the RMS status code is represented by RMSSTATUS%.

```
15000  DEF FN,GETSTA
15025  CALL FDV$STAT ( FMSSTATUSZ, RMSSTATUSZ )
```



Statement 15030 returns control to the statement following the GETSTA call should FMSSTATUS% reveal no error conditions. Otherwise, control passes to statement 15730, which uses the FDV\$DTERM call to detach the terminal.

```
15030  IF FMSSTATUSZ > 0 THEN FNEXIT
15730  CALL FDV$DTERM I TCAZ I
```

Statements 15735 to 15745 print the error codes in FMSSTATUS% and RMSSTATUS%. Statement 15747 contains the STOP instruction, and statement 15750 ends the status-checking function.

```
15735  PRINT "FDV ERROR. ' '
15740  PRINT " " , "FMS STATUS:" , FMSSTATUS%
15745  PRINT " " , "RMS STATUS:" , RMSSTATUS%
15747  STOP
15750  FNEND
15999  END
```

---

## Note

Although FMS supports the standard OpenVMS Run-Time Library error-reporting procedures, the procedure used for indicating an error condition in the Sample Application and the subset application does not conform to the OpenVMS conventions. For more details on error-reporting procedures, see the *VSI FMS Form Driver Reference Manual*

---

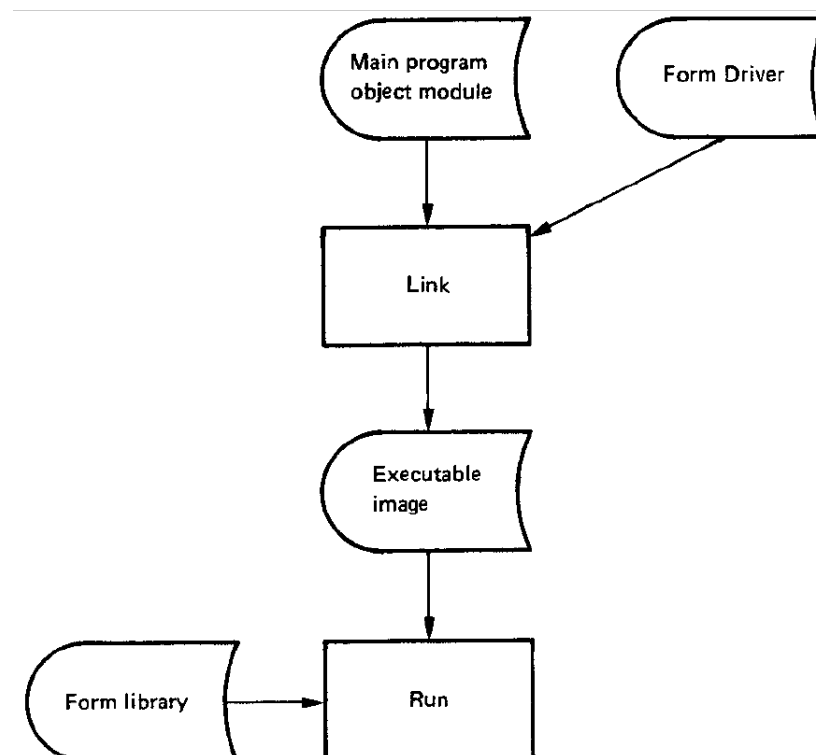
You have now entered all the statements for the subset application. *Chapter 6, "Compiling, Linking, and Running an Application"* tells you how to compile, link, and run this program.



# Chapter 6. Compiling, Linking, and Running an Application

As you read this chapter, you will learn how to create an executable image of the Sample Application subset that you prepared in *Chapter 5, "Writing an Application"*. *Figure 6.1, "FMS Application Development Cycle"* shows the steps in the FMS application development cycle that are covered in this chapter.

**Figure 6.1. FMS Application Development Cycle**



## 6.1. Compiling the Subset

To compile the subset application, type the following command:

```
$ BASIC SUBSET/LIST
```

If you want to create the subset application in a language other than VAX-11 BASIC, consult the *VSI FMS Language Interface Manual*.

## 6.2. Linking the Subset

Before you begin this step, check with your system manager to verify that the Form Driver library has been installed in the OpenVMS system library. If this has been done, type the following command to link the subset object module:

```
$ LINK SUBSET
```

By default, OpenVMS scans the system library to link the appropriate Form Driver library modules with the subset object module.

## 6.3. Running the Subset

To run the subset, type the following command:

```
$ RUN SUBSET
```

# Chapter 7. Programming Features

FMS can simplify your programming and can make forms more attractive and more useful. Four such features of FMS are discussed in this chapter:

- Indexing
- Scrolling
- Named Data
- User action routines

This chapter describes these features in detail and lets you see how they are used in the Sample Application subset. In this chapter, you will create a subroutine, named VUEREG, for the subset and a check register form, REGISTER, in which you will incorporate these features. You can then rebuild the subset application and verify that each feature performs correctly.

The *VSI FMS Form Driver Reference Manual* describes other programming features that FMS offers.

## 7.1. Indexing

Indexing simplifies creating and accessing fields that are identical. Indexed fields have the same name and attributes and are distinguished by indexes, much like an array. Your application can access these indexed fields in any order. Indexing also simplifies the procedure of assigning attributes to identical fields.

In the exercise that follows, you will learn about indexing by implementing it in the check register form, REGISTER. You will first be guided through the creation of REGISTER. You will then index the four fields in REGISTER.

The following image is the REGISTER form from SAMP, the form in which you can see a register of the checks written and deposits made.

### CHECK REGISTER - THE ACCOUNT HISTORY

Chk. No.	Date	Check Payee or Deposit Memo	Deposit Amount	Check Amount	New Balance
	15-MAR-82	Interest on National Coal bond	500.00	.	500.00
1	15-MAR-82	Jack Dewar	.	10.00	490.00
2	30-JUN-82	Louise Phipps	.	20.00	470.00
3	14-JUL-82	Townsend Fabrics	.	250.00	220.00
4	30-JUL-82	Channel 42	.	50.00	170.00
	31-AUG-82	Paycheck	300.00	.	470.00

This Session: Starting Balance: \$ 361.30  
 Total Deposits: \$ . 0  
 Total Checks: \$ . 0  
 Current Balance: \$ 361.30

To scroll through the check register, press UPARROW or DOWNARROW.  
 To return to the menu, press RETURN.

As you look at the form, note the following:

1. The title, CHECK REGISTER - THE ACCOUNT HISTORY, is centered on line 2 and is in double-wide characters.
2. The column heads for the scrolled area are located as follows:

<i>Chk. No.</i>	<i>Chk.</i> , line 5, column 2; <i>No.</i> , line 6, column 2
<i>Date</i>	Line 6, column 8
<i>Check Payee</i>	Line 6, column 17
<i>Deposit Amount</i>	<i>Deposit</i> , line 5, column 54; <i>Amount</i> , line 6, column 54
<i>Check Amount</i>	<i>Check</i> , line 5, column 63; <i>Amount</i> , line 6, column 63
<i>New Balance</i>	<i>New</i> , line 5, column 72; <i>Balance</i> , line 6, column 72

3. Horizontal lines, created by the DRAW function, extend from columns 1 to 79 on lines 7 and 14. Vertical lines extend from columns 1, 6, 16, 53, 62, 71, and 79 to the two horizontal lines.
4. The scrolled area has six lines. You will create this scrolled area in Section 7.2.
5. Below the scrolled area is a summary of the check register figures. *This Session:* begins in line 15, column 22. *Starting Balance:* begins in column 37. The field begins in column 56 and contains six characters and a decimal field-marker character. The field-validation character used in this field is 9, indicating that only numerals are to be accepted in this field. This field's attributes are Display Only, Right Justified, Zero Fill, Zero Suppress, and a zero (0) Clear Character.

6. The remaining summaries – *Total Deposits*, *Total Checks*, and *Current Balance*: – appear directly below, and the fields are identical to the field next to *Starting Balance*:. These summary fields are display only and will, along with the *Starting Balance* field, be indexed.
7. The background text at the bottom of the screen is in a box that occupies lines 20 to 23, from columns 14 to 80.

### 7.1.1. Create REGISTER

Invoke the Form Editor and create the form REGISTER, following the same steps you used in *Chapter 3, "Creating Forms"*.

1. From the menu, go to the Form phase. Assign a white screen background.
2. Go back to the menu and then go to the Layout phase. Now create the form according to the design just discussed. Leave the scrolled area blank for now. You will draw in the lines after you have created the scrolled area.
3. In the Assign phase, you assign indexing attributes to the summary fields in the middle of the form. The procedure for assigning these attributes follows.

### 7.1.2. Assign Indexing Attributes

Note that the four fields have identical attributes. Follow the steps below to index these fields:

1. As you assign attributes to the *Starting Balance* field, assign the name SUMMARY.
2. In the field next to *Index Value*, type 1, indicating the first in a set of indexed fields. Note that 1 appears in the blank below *Index Value*, captioned *of*. This blank shows how many fields are in the set of indexed fields.
3. Assign the Display Only, Right Justified, Zero Fill, Zero Suppress, and zero (0) Clear Character attributes.
4. Press RETURN to assign attributes to the next field.
5. For the field's name, again type SUMMARY. Assign the same attributes you assigned to the first field you created. Note that after you press TAB when you assign the field name, the *Index Value* and *Index Count* blanks are automatically updated. Both blanks contain the value 2. If you do not assign the same attributes that you assigned to the first field you created, the Form Editor signals you with an error if you press RETURN.
6. Repeat steps 4 and 5 until all four fields are indexed.
7. Return to the menu and type EXIT.

FMS provides an alternative to the procedure listed above:

1. While in the Layout phase, create the first field to be indexed.
2. Move the cursor into the field you just laid out and press GOLD FLDATR.
3. Assign the name, SUMMARY.
4. In the field next to *Index Value*, type 1.

Assign the Display Only, Right Justified, Zero Fill, Zero Suppress, and zero (0) Clear Character attributes.

5. Duplicate the field you just created four times, using the following procedure:

- Move the cursor to a character position in the *Summary* field, SUMMARY.
- Press CUT.
- Press GOLD PASTE to replace the field you just cut.
- Move the cursor to the location where you want to place the second indexed field: line 16, column 56.
- Press GOLD PASTE.
- Repeat the previous two steps until all four fields have been pasted into their appropriate locations. The Form Editor automatically updates their index values and assigns the same attributes assigned to the first field in the indexed set.

6. Return to the menu and exit.

When you are finished with the Layout, Form, and Assign phases of creating REGISTER, exit from the Form Editor.

### 7.1.3. Manipulate Indexed Fields in the VUEREGB Subroutine

In this section, you will write the statements required to manipulate the indexed fields. The statements in VUEREGB do the following:

1. Display the check register form, REGISTER, that you just created
2. Use the FDV\$PUT call to assign numeric values to the four indexed fields

---

#### Note

Make sure that you delete statements 13005, 13010, and 13015 that you have already entered for the VUEREGB sub routine. During run time, those statements display a message indicating that VUEREGB is not implemented.

---

In the VUEREGB subroutine, the values to be sent to the indexed fields have been established in the main body of the program. These values are the following arbitrary figures:

SBALANCE%	The starting balance: \$503.75
TOTPAY%	The total amount written in checks: \$213.45
TOTDEP%	The total amount of deposits made: \$1123.23
BALANCE%	The current balance

The statements you are about to write use the BASIC STR\$ function to make character strings of the values to be sent to the indexed fields.



Using an editor, type the following statements to display REGISTER and to assign values to the indexed summary fields:

```
13047 CALL FDV$CDISP ( 'REGISTER' )
13050 CALL FDV$PUT ( STR$ ( SBALANCE% ), 'SUMMARY', 1% )
13055 CALL FDV$PUT ( STR$ TOTDEP% ), 'SUMMARY', 2% )
13060 CALL FDV$PUT ( STR$ ( TOTPAY% ), 'SUMMARY' 3% )
13065 CALL FDV$PUT ( STR$ ( BALANCE% ) 'SUMMARY', 4% )
```

In the sections that follow in this chapter, you will be adding statements and features to VUEREg and to REGISTER.

## 7.2. Scrolling

In *Chapter 2, "Running the Sample Application Program"*, you saw how the check register form, REGISTER, in the Sample Application program used scrolling. Scrolling lets you display and edit a list of data that is too long to fit on the screen at one time.

The area of the form that displays the list of checks written is the scrolled area. When REGISTER first appears on the screen, the cursor is on the bottom line of the scrolled area. If you press DOWNLINE once, the list scrolls forward one line. The line that was at the top of the list disappears from the screen, and a new line appears at the bottom of the list. The cursor is still on the bottom line of the window.

If you press DOWNLINE several times, you reach the last item in the list. When the cursor reaches the end of the list, the following message appears at the bottom of the screen:

```
Last line of register
```

If you press UPLINE, the cursor moves up the list. Not until it reaches the top of the list does the list begin to scroll.

When a list of data appears in the scrolled area on a form, the Form Driver is aware only of the lines that appear on the screen. The application is responsible for moving the list of data as necessary.

The application must keep track of pointers to the following items to execute scrolling:

- The last item in the list of data
- The item that appears at the top of the scrolled area
- The bottom line in the scrolled area

Two Form Driver calls are used in the Sample Application to support scrolling:

FDV\$PFT	Process field terminator
FDV\$PUTSC	Output data to current line of scrolled area

You will now create the scrolled area in REGISTER and write the subroutine in the Sample Application that executes the scrolling.

### 7.2.1. Complete the Check Register Form, REGISTER

Before you continue with this exercise, make sure that you have completed the exercises in *Chapter 3, "Creating Forms"* and the indexing exercise in *Section 7.1.3, "Manipulate Indexed Fields in the VUEREg Subroutine"*.

First, examine the design of the scrolled area in the REGISTER form:

Chk. No.	Date	Check Payee or Deposit Memo	Deposit Amount	Check Amount	New Balance
	15-MAR-82	Interest on National Coal bond	500.00	.	500.00
1	15-MAR-82	Jack Dewar	.	10.00	490.00
2	30-JUN-82	Louise Phipps	.	20.00	470.00
3	14-JUL-82	Townsend Fabrics	.	250.00	220.00
4	30-JUL-82	Channel 42	.	50.00	170.00
	31-AUG-82	Paycheck	300.00	.	470.00

Note the characteristics of the scrolled area:

1. The scrolled area has six lines, and each line has six visible fields. Each field is display only.
2. The first field, NUMBER, begins in line 8, column 2, and can have up to four characters.
3. The second field, DATE, begins in column 7, but does not use the Date function. Instead, this field has the following field picture: XX-XXXXXX. If you were to use the Date function to create a field picture for this field, the Form Driver would display only the current date in this field, not the date representing the day of the transaction.
4. The third field, PAYMEM, has 35 characters, starting in column 17 and extending to column 51.
5. The fourth field, DEPOSIT, starts in column 54 and has six characters and a period field-marker character. This field has the Display Only, Right Justified, Zero Fill, Zero Suppress, and zero (0) Clear Character attributes.
6. The fifth field, AMTPAY, starts in column 63 and has the same characteristics as DEPOSIT. This field has the Display Only, Right Justified, Zero Fill, Zero Suppress, and zero (0) Clear Character attributes.
7. The sixth field, BALANCE, starts in column 72 and is identical to the DEPOSIT field. This field has the Display Only, Right Justified, Zero Fill, Zero Suppress, and zero (0) Clear Character attributes.

A seventh field exists in this form. Unlike the other fields, this field, located in column 80, is a dummy field. Scrolled lines should have at least one readable field – a field that is not display only – if you wish to place the cursor in each scrolled line. You can do this by placing a dummy field in each scrolled line. Each scrolled line, therefore, has this seventh field, named FAKE. It has the No Echo attribute and is never used by the application. FAKE provides only a means of placing the cursor in a scrolled line and of receiving a terminator for input.

All fields in the scrolled area use only the X field-validation character. Since all these fields, except FAKE, are display only, the X field-validation character is all that is necessary. FAKE also uses X; thus, during run time, the subset application ignores any characters that maybe entered.

When the Sample Application runs and REGISTER appears on the screen, the cursor is in this dummy field. The dummy field is the field from which the Form Driver retrieves the field terminator to begin scrolling.

To add a scrolled area to REGISTER, invoke the Form Editor and do the following:

1. Type LAYOUT to enter the Layout phase. Move the cursor to line 8.
2. Press GOLD FIELD.
3. Type in the field-validation characters for the field NUMBER, as noted in this form's design.
4. Lay out the remaining six fields, all with the X field-validation character. Remember that the field FAKE does not have a column head and that it consists of a single X field-validation character.
5. Press SCROLL.
6. Press DOWNLINE. Note that the last line you laid out is replicated.
7. Press DOWNLINE four more times until you have the six lines for the scrolled area.
8. Press ENDSCROLL.
9. With the cursor in any line of the scrolled area, try to change one of the fields. Note that the change is replicated in each line in the scrolled area. It is thus easy for you to edit the lines in the scrolled area.
10. When you are done with the scrolled area, draw in the lines around and through the scrolled area, as described in Section 1.1.
11. Exit from the Layout phase by pressing GOLD MENU to return to the menu.
12. Type ASSIGN to enter the Assign phase.
13. When in the Assign phase, note that you assign attributes to the fields in only one line of the scrolled area. Since all lines in the scrolled area must be identical, it is not necessary to assign attributes to the fields in every line in the scrolled area.
14. Press GOLD MENU to return to the menu and exit.

## 7.2.2. Insert REGISTER in SUBSET.FLB

You must now place REGISTER in the form library, SUBSET.FLB. To do this, use the following command:

```
$FMS/LIBRARY/INSERT SUBSET
```

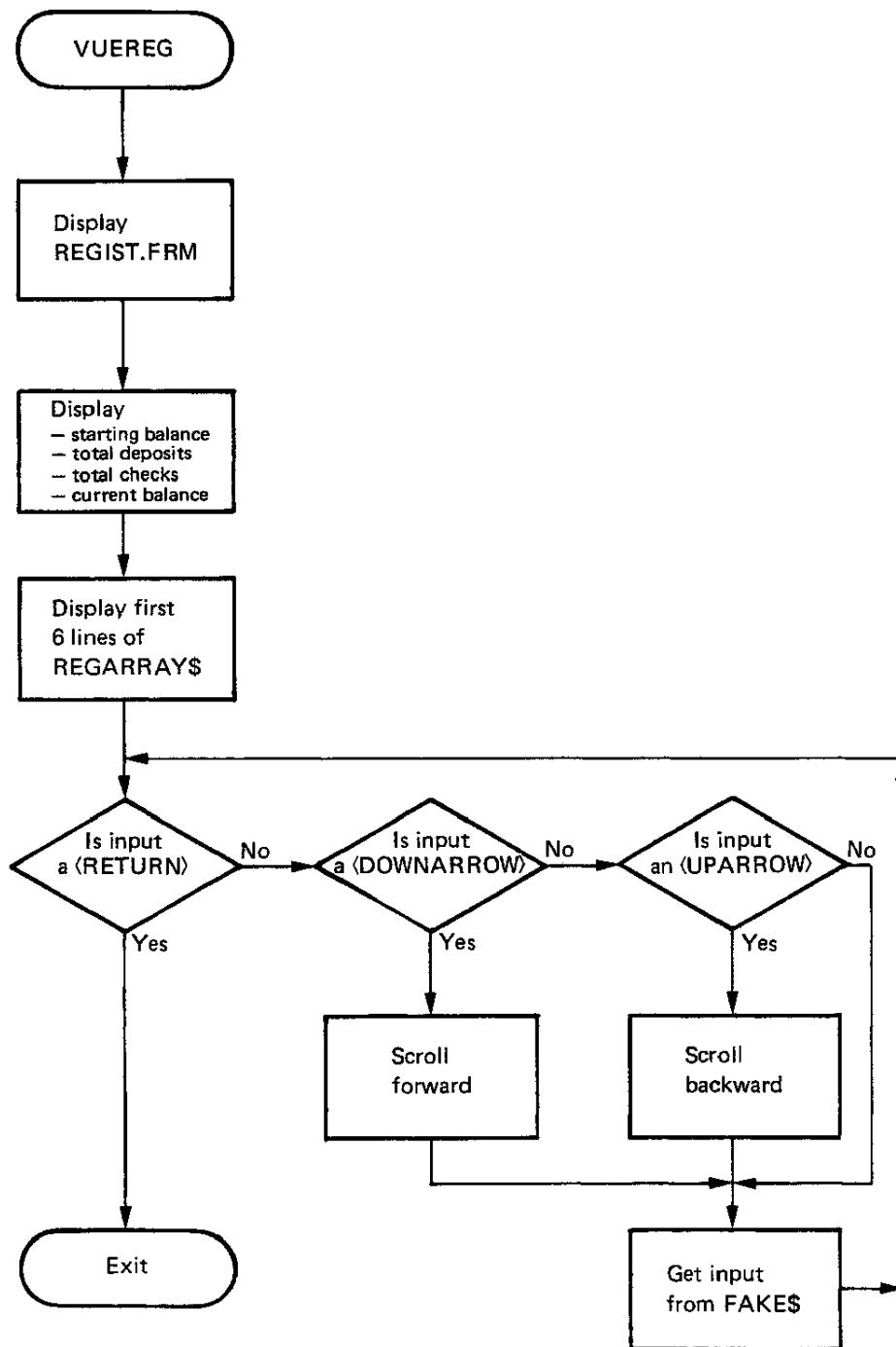
The Form Librarian responds with the following prompt:

```
_Input Files:
```

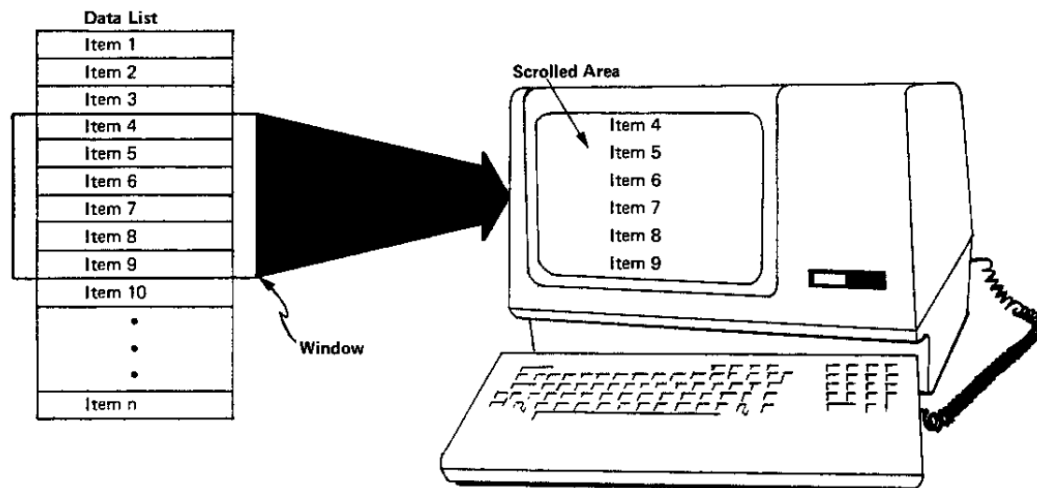
Type the name of the form you just created, REGISTER.

## 7.2.3. Writing the Statements that Support Scrolling

In this section, you will write the statements required to support scrolling in the check register form, REGISTER. Before examining the scrolling statements, look at the flow of control for the VUERE subrountine (see *Figure 7.1, "Flowchart for the VUERE Subroutine"*). The form REGISTER is called and displayed by the SAMP subrountine VUERE.

**Figure 7.1. Flowchart for the VUEREK Subroutine**

The data to be scrolled is in an array, REGARRAY\$. The application program, not the Form Driver, must keep track of which lines in REGARRAY\$ are in the scrolled area. The lines in REGARRAY\$ that are mapped to the scrolled area appear in a window (see *Figure 7.2, "Mapping a Window to a Scrolled Area"*). The process of scrolling can be thought of as mapping this window to the scrolled area on the terminal.

**Figure 7.2. Mapping a Window to a Scrolled Area****Note**

In the Sample Application, REGARRAY\$ is in a data file and must go through a series of processes before it can be scrolled on to a terminal screen. In the subset application, we will simplify REGARRAY\$'s data structure by assigning it 10 specific items in subroutine INACCT. Before continuing with this exercise, edit SUBSET.BAS and add the following statements to INACCT that make up REGARRAY\$. Use the DIM statement at the beginning of SUBSET.BAS to assign the array REGARRAY\$ a size of 10 bytes.

Edit SUBSET.BAS and make the following statement the first statement of the program:

```
100 DIM REGARRAY$ ( 10 )
```

Add the following statements to INACCT to compose make up array REGARRAY\$. (The numbers above the statements show the exact column positions of the data.)

1	9	26	69	76	82
4025	REGARRAY\$(1)=	15MAR82Interest on National Coal bond	050000		50000"
4030	REGARRAY\$(2)=	115MAR82Jack Dewar		1000	49000"
4035	REGARRAY\$(3)=	230JUN82Louise Phipps		2000	47000"
4040	REGARRAY\$(4)=	314JUL82Townsend Fabrics		25000	22000"
4045	REGARRAY\$(5)=	430JUL82Channel 42		5000	17000"
4050	REGARRAY\$(6)=	31AUG82Paycheck	030000		47000"
4055	REGARRAY\$(7)=	512SEP82Four-Star Auto		1543	45457"
4060	REGARRAY\$(8)=	6 40CT82Mary Johnson		4450	41007"
4065	REGARRAY\$(9)=	7 1FEB83Cory Advertising Agency		5002	36005"
4070	REGARRAY\$(10)=	04FEB83Pegasus Equestrian Center	000125		36130"
4075	LASTREGNUM%=10				

To perform scrolling, VUERE maps a section of REGARRAY\$ to the scrolled area.

**Display REGISTER**

The first five lines of the VUERE subroutine do the following:

- Declare integer constants for the keypad keys ENTER, DOWNLINE, and UPLINE.
- Issue the CDISP call to display the REGISTER form.
- Display the summary balance figures. (These statements appear in the exercise on indexing, but are repeated here.)

The first statement of VUEREG declares integer constants that represent the keypad keys the operator can press while REGISTER is displayed.

```
13000 DEF FN,VUEREG
13001 DECLARE INTEGER CONSTANT                                &
        FDV$K_FT_NTR = 0,          ! RETURN or ENTER        &
        FDV$K_FT_SFW = 8,          ! OOWNLINE (scroll forward) &
        FDV$K_FT_S6K = 9          ! UPLINE (scroll backward)
```

---

## Note

VAX-11 BASIC requires you to declare in the main program the integer constants representing each keypad field terminator. Some other languages that FMS supports permit you to use include or require files that contain these integer constants.

---

Enter the following lines to perform the initial display and setup of REGISTER. Note that the FDV\$PUT call uses indexing to display the summary figures.

```
13047 CALL FDV$CDISP ( 'REGISTER' )
13050 CALL FDV$PUT ( STR$( SBALANCE% ), 'SUMMARY', 1% )
13055 CALL FDV$PUT ( STR$( TOTDEP% ), 'SUMMARY', 2% )
13060 CALL FDV$PUT ( STR$( TOTPAY% ), 'SUMMARY', 3% )
13065 CALL FDV$PUT ( STR$( BALANCE% ), 'SUMMARY', 4% )
```

---

## Note

If you have completed the section on indexing, you do not need to enter the statements above.

---

The subroutine sets the variable NSCROL% to represent the number of lines in the scrolled area (6).

```
13085 NSCRDL% = 6%
```

## Initialize Two Scrolling Pointers

The next two statements establish pointers to two lines:

- A subscript in REGARRAY\$ corresponds to the top line in the scrolled area – MINWINDOW%. MINWINDOW% is initialized to 1.
- A subscript in REGARRAY\$ corresponds to the line the cursor is on – CURLINE%. CURLINE% is also initialized to 1.

```
13135 MINWINDOW% = 1
13110 CURLINE% = 1
```

## Output a Data Line into the Scrolled Area – FDV\$PUTSC

The FDV\$PUTSC call sends the current line of the window to the scrolled area on the screen. The first line from REGARRAY\$, the register array in the Sample Application, is sent to the first line in the scrolled area, using the FDV\$PUTSC call. When the FDV\$PUTSC call is issued, the entire line to be output is referenced by a single field name. This field can be any field in the line. The field also identifies which scrolled area to use, since a form can have more than one scrolled area. In this example, the line to be scrolled is referenced by the field NUMBER, the first field in each line in REGARRAY\$.

---

The FDV\$PUTSC call has the following format:

**FDV\$PUTSC ( fldnam, fldval )**

The argument **fldnam** represents a field name in the line to be scrolled, and **fldval** represents a variable name containing the value of the line to be scrolled.

Enter the following statement appearing in VUEREGL to send the first line in REGARRAY\$ to the scrolled area on the screen:

```
13115 CALL FDV$PUTSC ( 'NUMBER', REGARRAY$(1) )
```

**Output the Remaining Lines in the Scrolled Area**

The subsequent statements in VUEREGL occupy the remaining five lines in the scrolled area of the form REGISTER. Note the use of the variables MINWINDOW% and CURLINE%. Two additional variables are used in these statements:

MAXWINDOW%	A subscript of REGARRAY\$ that corresponds to the bottom line of the scrolled area
LASTREGNUM%	A subscript of the last item in REGARRAY\$

**Altering the Current Line - FDV\$PFT**

This section of the subroutine uses the FDV\$PFT call to increment the current line to the next line to be sent to the scrolled area.

The FDV\$PFT call has the following format:

**FDV\$PFT ( [fldtrm,] [fldnam,] [fldval,] [nfldnam,] [,nflidx] )**

The argument **fldtrm** represents the field terminator to be processed; **fldnam** is a field that identifies the scrolled area. The argument **fldval** represents the field values to be displayed if the screen is scrolled during the processing of the field terminator. The argument **nfldnam** represents the current field name after the call has been processed, and **nflidx** represents the index of the current field.

Enter the following statements to send the first six lines of REGARRAY% to the scrolled area in REGISTER. Note that the PFT call is used here to update the current line for the Form Driver and that the variable CURLINE% is incremented simultaneously for the application source code. The PUTSC call then sends the current line to the scrolled area.

```
13150 WHILE ( CURLINE%, LASTREGNUM% AND CURLINE% < NSCROL%
13155     CURLINEZ = CURLINEZ + 1
13160     CALL FDV$PFT ( FDV$K_FT_SFW, 'NUMBER' )
13165     CALL FDV$PUTSC ( 'NUMBER', REGARRAY$ ( CURLINE% ) )
13170 NEXT
13171 MAXWINDOW% = CURLINE%
```

**Scrolling Forward and Backward**

The rest of the subroutine does the scrolling by performing the following flow of control:

1. If the operator presses ENTER or RETURN, control returns to the menu.

2. If the operator presses DOWNLINE, the scrolled area scrolls forward.
3. If the operator presses UPLINE, the scrolled area scrolls backward.
4. If the operator presses any other key, the subroutine does nothing.

The subroutine uses the FDV\$GET call to get the operator input from the field FAKE\$. The scrolling operation needs only the field terminator, not any particular value, from FAKE\$. If the operator presses DOWNLINE, this routine branches to the SCRFWD subroutine. If the operator presses UPLINE, this routine branches to the SCRBAK subroutine,

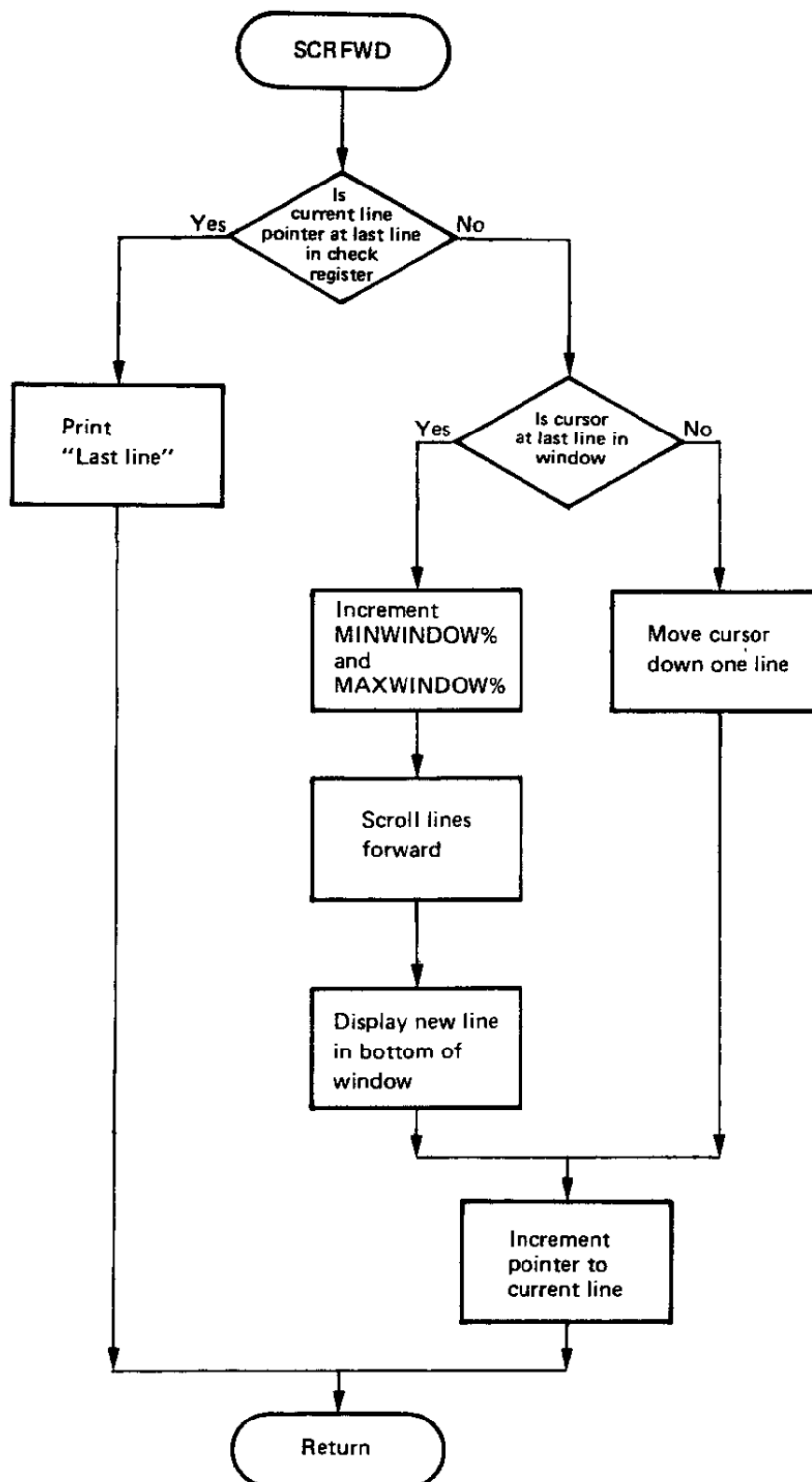
Enter the following statements, which make up the final section of VUEREG:

```
13215 CALL FDV$GET ( FAKE$, TERMINATOR%, 'FAKE' )
13220 WHILE NOT (TERMINATOR% = FDV$K_FT_NTR)
13225     IF TERMINATOR% = FDV$K_FT_SFW THEN C=FN, SCRFWD
13235     IF TERMINATOR% = FOV$K_FT_SBK THEN C=FN, SCRBAK
13245 CALL FDV$GET ( FAKE$, TERMINATOR%, 'FAKE' )
13250 NEXT
13255 FNEND
```

### **The Scroll Forward Subroutine – SCRFWD**

*Figure 7.3, "Flowchart for the SCRFWD Subroutine"* shows the flow of control in the SCRFWD subroutine.



**Figure 7.3. Flowchart for the SCRFWD Subroutine**

In summary, the SCRFWD subroutine works as follows:

1. Check to see if the current line is pointing to the last line in the check register. If it is, print "Last line of register" at the bottom of the screen and go back to the body of VUERG. If not, proceed with the subroutine.
2. If the cursor is not at the last line of the window, move the cursor down, increment CURLINE%, and return to VUERG. If the cursor is at the last line of the window:

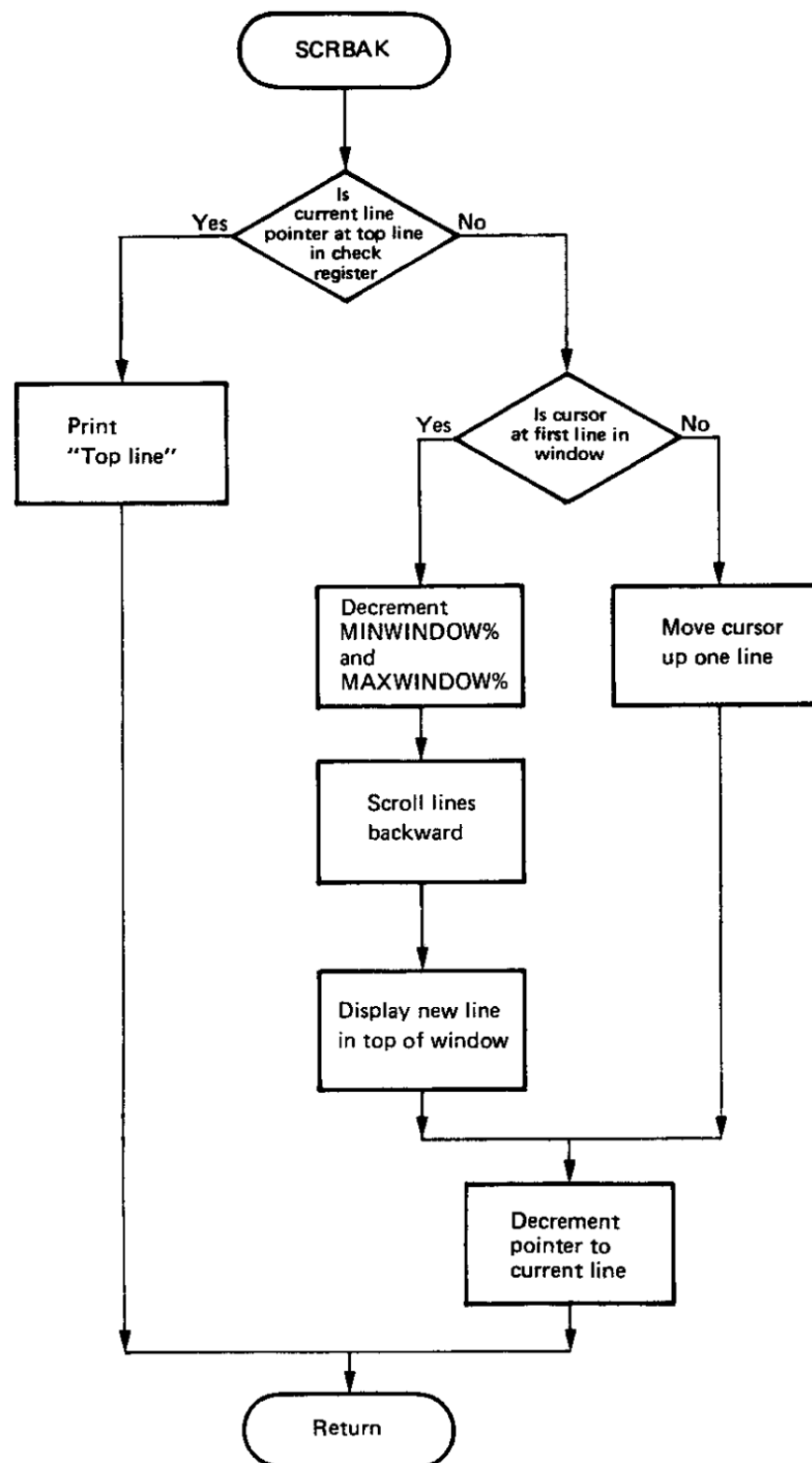
- Increment MINWINDOW% and MAXWINDOW%
- Scroll the lines in the window up one position and display a new line at the bottom of the window
- Increment CURLINE%
- Return to VUERE

Enter the following statements for the SCRFWD subroutine. Note the use of the FDV\$PFT call. If scrolling occurs, FDV\$PFT updates the current line and sends the new data line to the scrolled area. This special use of the FDV\$PFT call makes a FDV\$PUTSC call unnecessary in this routine.

```
13500 DEF FN, SCRFWD
13540 IF CURLINE% = LASTREGNUM% THEN
        CALL FDV$PUTL ( 'Last line of register' )
        FNEXIT
13580 IF CURLINE% <> MAXWIND% WZ THEN
        CALL FDV$PFT ( FDV$K_FT_SFW, 'NUMBER' )
        ELSE
            MINWINDOW% = MINWINDOW% + 1
            MAXWINDOW% = MAXWINDOW% + 1
            CALL FDV$PFT ( FDV$K_FT_SFW, 'NUMBER', REGARRAY$ ( MAXWINDOW% ) )
13585 CURLINE% = CURLINE% + 1
13590 FNEND
```

### **Scroll Backward Subroutine – SCRBAK**

The SCRBAK subroutine does the opposite of SCRFWD. *Figure 7.4, "Flowchart for the SCRBAK Subroutine"* shows the flow of control in the SCRBAK subroutine.

**Figure 7.4. Flowchart for the SCRBAK Subroutine**

The SCRBAK subroutine works as follows:

1. Check to see if the cursor is at the top line in the REGARRAY\$. If so, display the following message at the bottom of the screen and return to VUERE\$:

```
First line of register
```

2. If the cursor is not at the first line in the window, move it up. If the cursor is at the first line of the window, move the window back one line and write the new first line to the first line of the scrolled area. (This action can be executed in one step by the FDV\$PFT call.)
3. Move the current line pointer, represented by CURLINE%, back one line.

Enter the following statements for the SCRBAK subroutine:

```

13700  DEF FN, SCRBAK
13740  IF CURLINE% = 1 THEN
        CALL FDV$PUTL ( 'First line of register' )
        FNEXIT
13780  IF CURLINE% <> MINWINDOW% THEN
        CALL FDV$PFT ( FDV$K_FT_SBK, 'NUMBER' )
    ELSE
        MINWINDOW% = MINWINDOW% - 1
        MAXWINDOW% = MAXWINDOW% - 1
        CALL FDV$PFT ( FDV$K_FT_SBK, 'NUMBER', REGARRAY$ &
            ( MINWINDOOW% ) )
13785  CURLINE% = CURLINE% - 1
13790  FNEND

```

You have now completed the programming to implement scrolling in the subset application. Rebuild the subset application and verify that VUEREK performs correctly.

## 7.3. Named Data

Named Data provides a convenient way for you, the application programmer, to store program parameters with the form instead of coding them into the program. By using Named Data, you can add forms and change existing forms without having to change the application program.

Named Data is data that is associated with a form but does not appear on the operator's screen. Named Data exists in the workspace with the form. You associate Named Data to a form during the Form Editor Data phase. The data can be any string up to 80 characters and can be referenced by either name or index. In *Table 7.1, "Data with Corresponding Names and Indexes"*, for example, an application could access NSTRING\$ by referencing either VALI, the name, or 1%, the index.

**Table 7.1. Data with Corresponding Names and Indexes**

1	VAL1	NSTRING
2	VAL2	PSTRING
3	VAL3	XSTRING
4	OPTION	EXIT

Two calls retrieve Named Data from forms: FDV\$RETDN and FDV\$RETDI.

### FDV\$RETDN

The FDV\$RETDN call has the following format:

**FDV\$RETDN ( nmdnam, nmdval [,nmdidx] )**

The argument **nmdnam** represents the name by which the data is referenced, **nmdval** represents a variable into which the Named Data is stored, and **nmdidx** represents the index of the data that is being returned.

The following example retrieves from a form the value PSTRING\$ by the name VAL2:

```
CALL FDV$RETDN ( 'VAL%', PSTRING$ )
```

### **FDV\$RETDI**

The FDV\$RETDI call works the same way as the FDV\$RETDN call except that the data is referenced by index. The FDV\$RETDI call has the following format:

**FDV\$RETDI ( nmdidx, nmdval [,nmdnam] )**

The argument **nmdidx** represents the index for the data to be retrieved, **nmdval** represents the variable into which the data is stored, and **nmdnam** represents the name for the data to be retrieved.

The following example returns the value PSTRING\$ referenced by the index 2:

```
CALL FDV$RETDI ( 2%, PSTRING$ )
```

You will now add Named Data to the check register form REGISTER. You will then modify the subroutine VUEREGL to incorporate the use of Named Data.

The subroutine VUEREGL uses Named Data to determine how many lines are in the scrolled area in REGISTER and then sets up the scrolled area accordingly. It is possible to change the size of the scrolled area in REGISTER and to support scrolling for that form without changing the scrolling statements in the subroutine.

In this exercise, you will change the size of the scrolled area from six to five lines. Next, you will change the program to make the program pick up the size of the scrolled area from the form. Thus, later changes to the size of the scrolled area will not require changes to the program.

### **Modify REGISTER**

Extract REGISTER from the library SUBSET.FLB and then edit REGISTER. You can do this in the single following command:

```
$ FMS/EDIT SUBSET/FORM_NAME=REGISTER
```

The screen clears and the Form Editor menu appears. Type DATA to enter the Data phase and press RETURN. A questionnaire appears on the screen (see *Figure 7.5, "Assigning Named Data"*).

**Figure 7.5. Assigning Named Data**

```

Named Data

1  Name |-----
-----
2  Name -----
-----
3  Name -----
-----
4  Name -----
-----
5  Name -----
-----

```

The numbers 1 to 5 are numbers for the items. Each item has a space for a name and a value. The cursor is in the name blank for the first Named Data item. Type the name Window and then press TAB. With the cursor on the second line, type 5 and press RETURN (see *Figure 7.6, "Assigning Named Data"*).

**Figure 7.6. Assigning Named Data**

```

1  Name WINDOW-----
2  5-----

```

The menu appears on the screen. Type LAYOUT to enter the Layout phase. Press RETURN. REGISTER appears on the screen. Do the following:

1. Move the cursor to the last line in the scrolled area.
2. Press GOLD UNSCROL.
3. Press GOLD INSERT.
4. Press GOLD DELEOL.
5. Press GOLD MENU.
6. Type EXIT.

Now update the form library SUBSET.FLB so that it contains the modified form, REGISTER. Do this with the following command:

```
$FMS/LIBRARY/REPLACE SUBSET/FORM_NAME=REGISTER REGISTER
```

### Modify the VUEREK Subroutine

In VUEREK, the size of the scrolled area is established by the variable NSCROL%, which you assigned the value 6 during the exercise on scroll ing. Using Named Data, you will retrieve a value for NSCROL% from the Named Data section in REGISTER. Since the value you supplied in the Named Data section was 5, NSCROL% will be assigned that value.

Now you will edit VUEREK so that it uses the FDV\$RETDI call to retrieve the value 5 and stores it in the variable NSCROL%. Using an editor, modify VUEREK as follows:

1. Where NSCROL% previously was assigned the value 6, you first initialize NSCROL\$ to represent a blank. (VAX-11 BASIC requires you to pre-extend all string variables. In this case, you must create a single space for NSCROL\$ so that it can later be set to represent a string value.)
2. Add the FDV\$RETDI call to retrieve the value 5, which was set up to be referenced by the index 1 when you modified the register form, REGISTER. The value 5 will be stored in NSCROL\$.
3. Use the BASIC VAL function to give NSCROL% the numeric value 5.

Using an editor, go to the following line in VUEREK:

```
13085  NSCROL% = GZ
```

Delete that line and insert the following lines:

```
13085  NSCROL$ = ' '  
13090  CALL FDV$RETDI ( 1%, NSCRDL$ )  
13095  NSCROL% = VAL ( NSCROL$ )
```

Compile and link the subset one more time to test REGISTER and to see if the scrolled area has five lines.

## 7.4. User Action Routines

User action routines, or UARs, permit you to write routines that can be executed when the operator does one of the following:

- Completes a field
- Requests help
- Presses a special key

There are three types of UARs:

- Field completion
- Help
- Function key

UARs are called directly by the Form Driver. With UARs an application program can execute a procedure while the Form Driver has control of the processing. In this way, a UAR supplements the Form Driver.

A UAR is like any other subroutine except that it has no formal parameters associated with it. You associate a UAR with a form or a field during the Form Editor Assign phase. When you assign a UAR to a form or a field, you can specify a string of up to 80 characters to that UAR. The string acts as a parameter.

A UAR can exist in a file by itself, or it can be included in the main program source file. Typically, UARs that can be used in a variety of applications, called general-purpose UARs, are maintained in UAR libraries. UARs that are application-specific are commonly included with the main program. UARs can be compiled separately from the main program and need be bound with the FMS application only at link time. For more information on UAR libraries, see the *VSI FMS Form Driver Reference Manual*

### 7.4.1. Field Completion UARs

A field completion UAR is executed when a specific field in a form is completed by any FMS terminator other than BACKSPACE, the terminator that moves the cursor to the previous field. In the Sample Application, for example, a field completion UAR is used in the WRITE A CHECK form. When the operator enters the amount of the check and presses TAB, a UAR verifies that the check amount does not exceed the current balance. If the operator writes a check for more money than is in the account, the terminal beeps, and the video attributes for the *Current Balance* field change to bold and blinking.

If input to a field ends with the BACKSPACE terminator or with any function key that is returned to the application program, the field completion UAR is not called. Thus, the operator can always leave a field without having that field's associated UAR executed.

Another UAR in the Sample Application verifies that the operator enters a number in the range 1 to 5 for the choice in the menu. If the operator enters a number outside that range or a character, a message appears at the bottom of the screen, telling the operator to enter a number in the valid range.

### 7.4.2. Help UARs

You can use help UARs to provide an application-specific help facility. A pre-help UAR is activated when the operator presses HELP. Instead of processing a usual help request, the Form Driver passes control to the prehelp UAR. Pre-help UARs let the application take control of all help processing, field or form. Help UARs can also be useful in gathering statistics on how often help is requested for particular fields. Using help UARs, an application can provide more sophisticated on-line help.

A post-help UAR is called after any other Form Driver help processing. That is, after all help, field and form, has been given and just before the Form Driver is ready to display the "No Help Available" message, the Form Driver calls the post-help UAR.

### 7.4.3. Function Key UARs

You can use a function key UAR to perform a procedure when the operator types a specific key. In the Sample Application, for example, a function key UAR appears with the menu to make keypad keys 1 to 5 act like numeric keys and to verify that the operator has pressed no keys other than keypad 1, 2, 3, 4, 5, and keypad period.

### 7.4.4. Creating a Sample UAR

In this section, you will create a field completion UAR to verify input to a field. This UAR, entitled VALID1, is associated with the menu in the Sample Application. VALID1 checks to see if the input



to the menu falls within the valid range 1 to 5. If the input is outside that range, VALID1 displays a message at the bottom of the screen and accepts another number. If the input is valid, control returns to the application.

This exercise has five steps:

1. Modify the form MENU to associate the UAR VALID1 with the *Option* field.
2. Update the library file SUBSET.FLB so that it contains the modified menu form.
3. Write the programming statements that make up VALID1
4. Create a UAR vector module.
5. Use VALID1 to compile, link, and run the subset application.

### Modify the Menu Form

Extract MENU from the library SUBSET.FLB and edit it, using the following command:

```
$ FMS/EDIT SU6SET/FORM_NAME=MENU
```

UARs are associated with a field during the Assign phase, so type ASSIGN to enter the Assign phase.

When the Form Editor displays the first Assign phase questionnaire on the screen, type 3 to indicate that you wish to modify a specific field. When the Form Editor asks which field to modify, type OPTION.

After the Form Editor displays the Assign phase questionnaire, press TAB until the cursor is in the field next to *UARs?* Type Y. (See *Figure 7.7, "Assigning User Action Routines"*.)

**Figure 7.7. Assigning User Action Routines**

```

- Autotab      - Right Justify  - Uppercase
- No Echo      - Fixed Decimal  - Must Fill
- Display Only - Zero Fill     - Response Required Clear Character _
-              - Zero Suppress - Supervisor Only  UARs? (Y,N)  Y
  
```

The Form Editor responds by clearing the lower half of the screen and displaying a blank for naming UARs and for providing data strings for each UAR. Since VALID1 is the UAR you wish to associate with this form, type VALID1 next to *UAR Name:* (see *Figure 7.8, "Assigning User Action Routines"*).

**Figure 7.8. Assigning User Action Routines**

```

Checking Account Menu
Choose Option (1-5): 
    1 Exit
    2 Write a check
    3 Make a deposit
Field Completion User Action Routines
Field Name:  OPTION
1  UAR Name: VALID1
   Associated Data:
2  UAR Name:
   Associated Data:

```

Press TAB to move the cursor to the *Associated Data* field. Type the following:

```
12345
```

The string 12345 contains valid character for the *Option* field. You will see how this string is manipulated in the statements that make up VALID1.

Press GOLD MENU to return to the Form Editor menu. Then type EXIT.

### Update SUBSET.FLB

Update the library file SUBSET.FLB so that it contains the modified form MENU. Type the following command:

```
$ FMS/LIBRARY/REPLACE SUBSET/FORM_NAME=MENU MENU
```

Having associated a UAR with a form, you are now ready to write the statements that make up the body of the UAR.

### Write the Statements for VALID1

The subroutine VALID1 can be compiled independently of the subset application. Since VALID1 is a general-purpose UAR, create it as an individual file, naming it VALID1.BAS. After it is compiled, you can store it in a UAR object library for future use. Details on creating and using UAR object libraries are supplied in the *VSI FMS Form Driver Reference Manual*.

Note that VALID1, like many other UARs in the Sample Application, is a general-purpose UAR. That is, you can use it in any FMS application to perform field validation of a single-character field without having to change the UAR. The only modification that must be made to use VALID1 is in the Form Editor Assign phase when you specify the associated data string.

The first statement in VALID1 establishes the codes the Form Driver uses to determine the results of the UAR upon completion.

Enter the following statements.

```
16005  FUNCTION INTEGER VALID!
16089  DECLARE INTEGER CONSTANT                                &
        FDV$K_UVAL_SUC = 1000, !Field completion success &
        FOV$K_UVAL_FAIL = 1001 !Field completion failure
```

The next four statements in VALID1 pre-extend the string variables into which FMS will return special values. Note that pre-extending string variables is a requirement of the VAX-11 BASIC language, not necessarily of other languages. The four string variables used in VALID1 are listed and described below.

FRMNAME\$	The name of the form in which the single-character field is used. In the Sample Application, the form is MENU.
UARVAL\$	A string that contains all the valid entries into the field in the menu. In the Sample Application, this string is 12345.
FLDNAME\$	The name of the single-character field.
FVALUE\$	The single-character value the operator has entered into the field.

Enter the following statements, which use the BASIC SPACE\$ function to pre-extend the string variables.

```
16096  FRMNAME$ = SPACE$(31)
16097  UARVAL$ = SPACE$(80)
16098  FLDNAME$ = SPACE$(31)
16099  FVALUE$ = SPACE$(1)
```

The next statement uses the FDV\$RETCX call to return the current context of the Form Driver to VALID1. The FDV\$RETCX call returns the following values to VALID1:

TCA%	Address of the terminal control area
WKSP%	Address of the workspace
FRMNAME\$	Name of the current form
UARVAL\$	Character string associated with this UAR
CURPOS%	Cursor position within the current field
FLDTRM%	Field terminator used
INSOVR%	Character input mode (Insert or Overstrike)
HELPNUM%	Number of times HELP has been pressed for the current field (nonzero only in a help UAR)

Although VALID1 returns all these values with the FDV\$RETCX call, only UARVAL\$, the variable that represents the UAR's associated data, is used. The remaining values are ignored by this UAR.

Enter the following statement to return special information to the UAR:

```
16140  CALL FDV$RETCX ( TCA%, WKSP%, FRMNAME$, UARVAL$, &
        CURPOS%, FLDTRM%, INSOVR%, HELPNUM% )
```

Enter the following statement to retrieve the current field name and index.

```
16145 CALL FDV$RETFN ( FLDNAME$, FINDEX% )
```

Using the name and index just retrieved by the FDV\$RETFN call, the next statement uses the FDV\$RET call to retrieve the value that the operator has entered into that field.

Enter the following statement:

```
16150 CALL FOV$RET ( FVALUE$, FLDNAME$, FINDEX% )
```

After issuing the FDV\$RETCX, FDV\$RETFN, and FDV\$RET calls, VALID1 checks to see if the operator input, represented by FVALUE\$, occurs in the string UARVAL\$. VALID1 uses the VAX-11 BASIC function POS to do this.

Enter the following statements:

```
16185 IF POS( UARVAL$, FVALUE$, 1) > 0 THEN
      VALID1 = FDV$K_LJVAL_SUC      !Success
    ELSE
      CALL FDV$PUTL* 'Illegal value' )
      VALID1 = FDV$K_UVAL_FAIL
16210 FUNCTIDNEND
```

### Create a UAR Vector Module

The UAR vector module is a file, existing in object file format, that provides the necessary information to the linker to associate VALID1 to the menu form, MENU. When you create the UAR vector module, assign it the name SUBSTVCTR, using the following command:

```
$FMS/VECTOR SUBSET/OUTPUT=SUBSTVCTR
```

This command causes FMS to scan SUBSET.FLB for any occurrences of UARs. When FMS finds VALID1, FMS sets up a data structure that enables VALID1 to be called at the appropriate time.

### Compile, Link, and Run

Using VALID1 to execute the subset application, first compile the UAR, using the following command:

```
$ BASIC VALID1
```

To link the subset application with VALID1, you must include the name of the UAR vector module, SUBSTVCTR. Type the following command:

```
$ LINK SUBSET, VALID1, SUBSTVCTR
```

After the subset is linked, run the application and verify that VALID1 performs correctly. When the SUBSET menu appears on the screen, type a number outside the range 1 to 5. When you press RETURN, the following message should be displayed at the bottom of the screen:

```
Illegal value
```

A field completion UAR returns one of the three values listed below to the Form Driver each time the operator enters a value into the field with which the UAR is associated (in this case, the *Option* field in MENU).

FDV\$K_UVAL_SUC	The value entered by the operator was valid. The Form Driver should now call any additional UARs
-----------------	--

	for this field. If the operator input is also found valid by any additional UARs, the operator input for this field is accepted.
FDV\$K_UVAL_END	The value entered by the operator was valid. The Form Driver should not call any additional UARs for this field but should accept the operator input immediately.
FDV\$K_UVAL_FAIL	The value entered by the operator was invalid. The Form Driver should signal the operator and request that input be reentered.

When you run the subset application, VALID1 checks to see if your input to the menu is valid. If it is, VALID1 returns FDV\$K\_UVAL\_SUC to the Form Driver, stating that the value entered was valid. Otherwise, VALID1 displays the "Illegal value" message and returns FDV\$K\_UVAL\_FAIL, telling the Form Driver to signal the operator and request that another value be entered.



# Chapter 8. Advice to New Users

This chapter presents information that you may find useful when designing forms. *Section 8.1, "Good Form Design"* discusses elements of good form design. The text provides many helpful hints and techniques to use when designing forms.

*Section 8.2, "Use of the Video Screen"* explains how to use the video screen as a human interface. The primary goal of creating a good human interface in forms is to lower the operator error rate. Following the guidelines presented in this chapter can help you achieve this goal.

Although this chapter provides many techniques and guidelines, this information is not the final word on form design. This chapter merely offers advice.

## 8.1. Good Form Design

The many books available on form design can give you good ideas for your FMS applications. Many of the ideas on preparing paper forms can be applied to video forms. This section highlights principles of designing successful forms. Good form design consists of two goals:

- To make forms that are easy to use
- To lower the operator's input error rate

When designing forms, keep in mind the following:

- Sort the information you wish to gather
- Provide a good title for the form
- Select good captions
- Use check boxes
- Use reverse video as a visual aid
- Make the form self-instructing

The sections that follow discuss these techniques in detail.

### 8.1.1. Sorting Information

Before you sit down at the terminal to design a form, write down all the information that you wish to gather and to display in the forms. This information can be grouped into the following categories:

- Key information (name, address, identification number, and so forth)
- Instructions
- Tables, lists, and supplemental data

Next, approximate the length of each item appearing on the form. This list should tell how long the caption and the desired response should be. For example, an address usually requires three lines of about 40 characters each.

The key information should appear in a conspicuous place on the forms – at the top center or top left, for example. Since such information is often used to identify a form, it needs to be readily visible.

Decide which instructions to include on the form and which to put in help forms. Generally, self-instructing forms are more successful than those requiring separate instructions.

Group information in sections of a form or on a separate forms. Use lines and boxes to separate groups of information that appears on the same form. *Figure 8.1, "Sample Payroll Data"* shows a rough grouping of all the information required for a series of forms for a payroll application.

**Figure 8.1. Sample Payroll Data**

<b>Employee Data</b>	<b>Salary Information</b>
Home Address	Wage Class
Home Phone	Salary Group
Badge Number	Current Salary
Social Security	Deductions
Date of Hire	
Dependents	
<b>Emergency Contact</b>	<b>Supervisor Information</b>
Name	Name
Address	Phone Number
Phone Number	
<b>Department Information</b>	<b>Job Information</b>
Name	Title
Number	Number
Location	Description

## 8.1.2. Providing a Title

A title should be as descriptive as possible of the purpose of the form. The location of the form title is a matter of personal choice. Top center is the usual position, although top left is also common. If a company name and address appear on the form, place them below the title. Highlight the title with a video attribute, such as bold or reverse, or use double-size or doublewide characters.

Avoid using the word form in a title; it adds little to meaning.

## 8.1.3. Writing Good Captions

When making up a form, use care in writing captions. Good captions can help to minimize data entry errors. It is important that captions and other words on a form be easy to understand. The few words you use in a caption must leave no doubt as to the desired response. Good captions result in:

- Better answers, in less time
- Less need for instructions and help
- Easier maintenance of the form

For example, in a date field, specify which date the operator is to enter: today's date, effective date, termination date, delivery date, and so forth.

## 8.1.4. Using Check Boxes

Captions and check boxes can be set up to provide multiple-choice and yes/no sections on your forms. Using check boxes, or single-character fields, simplifies data entry and reduces the data entry error rate by providing answers from which the operator can select an appropriate response.



When you have determined which answers are to have check boxes, you can then decide how to arrange them. The following are several ways to arrange the answers for a question on marital status.

**Marital status**    **Single** ☐    **Married** ☐    **Single, head of household** ☐  
    **Married, but spouse filing separately** ☐

**Marital status**    **Single** ☐  
    **Married** ☐  
    **Single, head of household** ☐  
    **Married, but spouse filing separately** ☐

**Marital status**    ☐ **Single**  
    ☐ **Single, head of household**  
    ☐ **Married**  
    ☐ **Married, but spouse filing separately**

<b>Marital status</b>			
<input type="checkbox"/> <b>Single</b>	<input type="checkbox"/> <b>Single, head of household</b>	<input type="checkbox"/> <b>Married</b>	<input type="checkbox"/> <b>Married, but spouse filing separately</b>

The last two examples are easier to read and respond to than are the first two.

With multiple-choice questions, check boxes should appear before the responses. Indicate which character the operator should enter in the check box, such as an X. With yes/no formats, check boxes should appear after the responses, as follows:

**Married** ☐  
**Employed** ☐  
**U.S. citizen** ☐  
**Head of household** ☐

## 8.1.5. Using Reverse Video Screen Characteristics

Reversing the screen characteristics can help to guide the operator's eyes across a form. Reverse video can:

- Separate sections of the form
- Highlight sections of the form
- Separate captions from fields
- Give the illusion of a colored background
- Give the form a pleasing appearance

Note that VT100 terminals with the advanced video option provide three intensities with which to work: bold, light, and reverse video.

## 8.1.6. Providing Instructions

Designing a form carefully reduces the need for lengthy instructions on how to fill it out. The only instructions necessary should say "Fill out Form XYZ." If more instructions are necessary, make them easy to understand.

Just as with captions, instructions should be easy to understand. Make sure that you explain the proper use of the form and describe its purpose clearly. Be careful not to over-explain simple items.

Keep in mind that many operators do not read instructions carefully. For that reason, you should format and word instructions so that they can be scanned easily. In the example below, the instructions in list form are easier to follow than are those in paragraph form.

Read the box at the top left of the screen before completing your answers. Answer all questions. Indicate yes or no responses where applicable. Press HELP if necessary.

1. Read the box at the top left of the screen before completing your answers.
2. Answer all questions.
3. Indicate yes or no responses where applicable.
4. Press HELP if necessary.

Instructions are most effective when they are close to the field or section to which they refer. If the captions are well chosen, few individual fields will need much explanation.

## 8.2. Use of the Video Screen

You must take special considerations into account when designing forms for video display. This section presents techniques used in some of the more successful video displays. Specifically, this section discusses:

- Data presentation
- Screen layout
- Form content
- Recovery procedures

### 8.2.1. How to Present Data

The section presents hints on presenting nonverbal information on a screen. Nonverbal information includes mostly numerical information, such as telephone numbers, part numbers, storage dumps, and so on. This section provides guidelines on presenting the following types of data:

- Lengthy alphanumeric strings
- Ordered data
- Data lists
- Punctuated data

#### **Lengthy Alphanumeric Strings**

Break down strings of alphanumeric characters into groups of three or four characters.

##### **Before**

```
125ACA6740B45C00  
5628CJVTH4838TND  
ABBA34568794FV10
```

##### **After**

```
125A CA67 40B4 5C00
5628 CJVT H483 8TND
ABBA 3456 8794 FV10
```

The FMS field-marker character for placing blanks in a field picture is B.

### Ordered Data

Organize data so that it can be easily read and identified. In the example that follows, the numbers are arranged by size, from left to right across the screen.

#### Before

```
12.4  15.34  3.1415
2.2   23.67  4.5   7.9
34.2  82.1   12.34
```

#### After

```
    2.2      3.1415      4.5
    7.9     12.34      12.4
15.34    23.67     34.2
82.1
```

### Data Lists

When possible, list the data vertically. Use numbered lists only when you are listing items that can be selected. Do not use letters to alphabetize lists. Begin numbered lists with 1, not 0. As shown in the example below, names or words should be left justified; numeric lists should be right justified.

1. Magtape
2. Cassette
3. Diskette
4. Cartridge

When listing numbers with decimal points, the decimal points should align.

```
    234.13
34214.45
    2.02
    103.89
```

With FMS you can use left- or right-justified fields to make the alignment easy.

Indent subordinate items in vertical lists.

```
MENU
  WRITE A CHECK
  MAKE A DEPOSIT
  VIEW CHECK REGISTER
  EXIT
WRITECHECK
DEPOSIT
```

When providing items from which the reader is to make a selection, arrange the items in alphabetical order if the list has more than seven items. If the list has seven or fewer items, put the more probable choices at the top of the list.

```
Choose a language
```

BASIC  
FORTRAN  
COBOL  
PASCAL  
APL  
LISP  
ALGOL

### **Punctuated Data**

Avoid unnecessary punctuation. Use periods after item selection numbers and sentences. Abbreviations, mnemonics, and acronyms should not include periods. Avoid using quotes to highlight words. Avoid hyphenating words.

## **8.2.2. Screen Layout**

This section presents guidelines on organizing groups of data on a video screen.

One goal of organizing information on a screen is to provide a structure that the operator can identify. When an operator can identify a specific structure or a pattern in a set of forms, the forms are generally easier to understand and work with.

Be consistent in your presentation of forms. One section of the form may always be used for instructions. Another section might be reserved for the title, and another section might contain fields for operator input. Establish a meaningful structure to a form, and use that structure in subsequent forms in the same application.

You can use reverse video in portions of a form or set off sections with lines. Do not break the screen up into too many small sections, however, or the form will be cluttered and confusing.

Use the following guidelines when formatting your information:

- Separate paragraphs by at least one blank line.
- When listing descriptions, advantages, alternatives, and so on, start each point on a new line. Put bullets (lowercase o) or hyphens before each point to make the list visually appealing.

With the Sample Application, you can

- Write checks
- Make deposits
- View the check register
- View the account data

Experiment with the special graphics available in the RULE character set for different types of bullets.

- In numbered lists, separate the number from the item by at least one space.
- Limit the user's choice to seven items for any one field.

## **8.2.3. Communication with the Operator**

Feedback is critical in any type of communication. In an interactive video session, timely feedback is essential for the operator to feel confident with the application. An operator can become very frustrated

with a program that fails to provide feedback about whether the program has been accepting any input. To assure the operator that input has been accepted as intended, make sure that your application provides meaningful, informative messages. This section discusses two ways of providing operator feedback: highlighting and messages.

### Highlighting

Highlighting the field into which the operator is entering input is an effective way of showing which portion of the form is being worked on. FMS provides several attributes that can be used to highlight sections of forms. These attributes are discussed in *Section 8.3, "FMS Field Attributes"*.

### Messages

Messages can prompt for more information, tell the operator what else to do, and diagnose an error condition. Messages should be concise and clear, stating only what the operator needs to know to continue. Also, messages should be positive rather than negative. For example, instead of saying "Illegal value," the message should say "Value must be in range 1 to 5."

## 8.2.4. Recovery Procedures

From an operator's viewpoint, being able to correct an error without having to go back to the beginning is most desirable in any application. Although feedback is necessary for operator recovery from an error, it is not always sufficient. Error messages try to tell how to recover from an error condition, but they cannot always direct the operator to the spot where the error occurred.

Recovery procedures should detect an error condition as soon as possible. User action routines that check for correct input and return operators to locations where errors occurred provide an effective means of recovery. When preparing your application programs, try to incorporate as many thorough recovery techniques as possible.

## 8.3. FMS Field Attributes

The FMS field attributes are listed below.

Autotab	The Form Driver is to consider a field complete when the operator types the last character in the current field. Depending on the Form Driver call issued by the application program, the cursor moves immediately to the next field without requiring the operator to press a field terminator key.
Clear Character	A character that appears in unfilled data positions in a field. A blank is the default.
Display Only	A field can display data, but the terminal operator cannot enter data.
Fixed Decimal	A field is all signed or all unsigned numeric and has a single decimal point (.) or a comma (,) as a field marker character.
Must Fill	Each character position in a field must be filled if any character is entered. This attribute is particularly useful with data such as telephone

	numbers and identification numbers, in which only a specific number of digits entered is meaningful.
No Echo	Input from the operator is not displayed on the screen. This attribute is most frequently used to keep passwords private.
Response Required	The operator must enter valid data into the field.
Right Justify	The data entered is to be aligned at the right; the remainder of the field may be filled with leading zeros or spaces.
Supervisor Only	A field is considered to be display only when this attribute is enabled for a terminal. Enabling and disabling supervisor-only mode is done by the application at run time.
Uppercase	All alphabetic characters entered in a field are displayed as uppercase characters and are also returned to the application as such.
Zero Fill	The field positions that do not contain data are returned to the program as zeros.
Zero Suppress	The leading zeros of a value entered in a right-justified or a fixed-decimal field are to be replaced with blanks.

# Appendix A. Subset Application Listing

```

130   DIM WORKSPACE% ( 3 )
140   DIM TCA% ( 3 )
1040  CALL FDV$ATERM ( TCA% (), 12%, 2% )
1042  CALL FDV$AWKSP ( WORKSPACE% (), 2000% )\C=FN.GETSTA
1050  CALL FDV$LOPEN ( 'SUBSET', 1% ) \C=FN.GETSTA
1100  !
1110  ! MAIN MODULE
1115  C = FN.INACCT !INITIALIZE ACCOUNT DATA
1170  C = FN.MENU PROCESS MENU REQUESTS
1180  !
1190  !CLOSING PROCEDURE
1200  CALL FDV$CLOS
1208  CALL FDV$DTERM( TCA%() )
1220  GOTO 15999
1230  !
4000  DEF FN.INACCT
4005  SBALANCE% = 50375
4010  TOTPAY% = 21345
4015  TOTDEP% = 112323
4020  BALANCE% = 141353
4025  REGARRAY$(1)=' 15MAR82Interest on National Coal bond 050000
    50000'
4030  REGARRAY$(2)=' 115MAR82Jack Dewar                      1000
    49000'
4035  REGARRAY$(3)=' 230JUN82Louise Phipps                    2000
    47000'
4040  REGARRAY$(4)=' 314JUL82Townsend Fabrics                 25000
    22000'
4045  REGARRAY$(5)=' 430JUL82Channel 42                       5000
    17000'
4050  REGARRAY$(6)=' 31AUG82Paycheck                          030000
    47000'
4055  REGARRAY$(7)=' 512SEP82Four-Star Auto                   1543
    45457'
4060  REGARRAY$(8)=' 6 4OCT82Mary Johnson                     4450
    41007'
4065  REGARRAY$(9)=' 7 1FEB83Cory Advertising Agency          5002
    36005'
4070  REGARRAY$(10)=' 04FEB83Pegasus Equestrian Center        000125
    36130'
4075  LASTREGNUM%=10
4080  FNEND
4085  !
5000  DEF.FNMENU
5005  !MENU CHOICES
5010  ! 1 => EXIT
5015  ! 2 => WRITE A CHECK
5020  ! 3 => MAKE A DEPOSIT
5025  ! 4 => VIEW CHECK REGISTER
5030  ! 5 => SHOW ACCOUNT DATA
5040  OPTION$ = ' '

```

```

5045 WHILE 1 = 1
5050     CALL FDV$CDISP( 'MENU' ) \C=FN.GETSTA
5070     CALL FDV$GET( OPTION$, TERMINATOR%, 'OPTION' ) \C=FN.GETSTA
5075     ON VAL( OPTION$ ) GOTO 5082, 5090, 5100, 5110, 5120
5081     ! OPTION 1: EXIT
5082     FNEXIT
5085     ! OPTION 2: WRITE CHECKS
5090     C = FN.WRITCH \GOTO 5130
5095     ! OPTION 3: MAKE A DEPOSIT
5100     C = FN.MAKEDEP \GOTO 5130
5105     ! OPTION 4: VIEW CHECK REGISTER
5110     C = VUEREGET \GOTO 5130
5115     ! OPTION 5: VIEW ACCOUNT DATA
5120     C = FN.VUEACT \GOTO 5130
5130 NEXT
5140 FNEND
5150 !
11000 DEF FN.WRITCH
11005 CALL FDV$PUTL( 'Write a check - not implemented yet' )
11010 CALL FDV$WAIT
11015 FNEND
11020 !
12000 DEF FN.MAKDEP
12050 CALL FDV$CDISP( 'DEPOSIT' ) \C=FN.GETSTA
12065 CALL FDV$PUT( STR$( BALANCE% ), 'CURBAL' ) \C=FN.GETSTA
12125 DEP.AMT$ = SPACE$(6)
12130 DEP.MEMO$ = SPACE$(35)
12135 CALL FDV$GET( DEP.AMT$, TERMINATOR%, 'DEPOSIT' )
12140 CALL FDV$GET( DEP.MEMO$, TERMINATOR%, 'MEMO' )
12155 BALANCE% = BALANCE% + VAL( DEP.AMT$ )
12160 CALL FDV$PUT( STR$( BALANCE% ), 'NEWBAL' ) \C=FN.GETSTA
12165 CALL FDV$PUTL( 'Deposit made - press RETURN or ENTER to continue' )
12170 CALL FDV$WAIT
12175 FNEND
12180 !
13000 DEF FN.VUEREGET
13001 DECLARE INTEGER CONSTANT &
        FDV$K_FT_NTR = 0, &
        FDV$K_FT_SFW = 8, &
        FDV$K_FT_SBK = 9
13047 CALL FDV$CDISP( 'REGISTER' )
13050 CALL FDV$PUT( STR$( SBALANCE% ), 'SUMMARY', 1% )
13055 CALL FDV$PUT( STR$( TOTDEP% ), 'SUMMARY', 2% )
13060 CALL FDV$PUT( STR$( TOTPAY% ), 'SUMMARY', 3% )
13065 CALL FDV$PUT( STR$( BALANCE% ), 'SUMMARY', 4% )
13085 NSCROL$ = ' '
13090 CALL FDV$RETDI ( 1%, NSCROL$ )
13095 NSCROL% = VAL ( NSCROL$ )
13135 MINWINDOW% = 1
13140 CURLINE% = 1
13145 CALL FDV$PUTSC( 'NUMBER', REGARRAY$(1) )
13150 WHILE ( CURLINE% < LASTREGNUM% AND CURLINE% < NSCROL% )
13155     CURLINE% = CURLINE% + 1
13160     CALL FDV$PFT( FDV$K_FT_SFW, 'NUMBER' )
13165     CALL FDV$PUTSC( 'NUMBER', REGARRAY$( CURLINE% ) )
13170 NEXT
13171 MAXWINDOW% = CURLINE%
13215 CALL FDV$GET( FAKE$, TERMINATOR%, 'FAKE' )

```



```

13220 WHILE NOT (TERMINATOR% = FDV$_FT_NTR)
13225     IF TERMINATOR% = FDV$_K_FT_SFW THEN C = FN.SCRFWD
13235     IF TERMINATOR% = FDV$_K_FT_SBK THEN C = FN.SCRBAK
13245     CALL FDV$GET( FAKE$, TERMINATOR%, 'FAKE' )
13250 NEXT
13255 FNEND
13315 !
13500 DEF FN.SCRFWD
13540 IF CURLINE% = LASTREGNUM% THEN
        CALLFDV$PUTL( 'Last line of register' )
        FNEXIT
13580 IF CURLINE% <> MAXWINDOW% THEN
        CALL FDV$PFT( FDV$_K_FT_SFW, 'NUMBER' )
    ELSE
        MINWINDOW% = MINWINDOW% + 1
        MAXWINDOW% = MAXWINDOW% + 1
        CALL FDV$PFT( FDV$_K_FT_SFW, 'NUMBER', REGARRAY$( MAXWINDOW% ) )
13585 CURLINE% = CURLINE% + 1
13590 FNEND
13595 !
13700 DEF FN.SCRBAK
13740 IF CURLINE% = 1 THEN
        CALL FDV$PUTL( 'First line of register' )
        FNEXIT
13780 IF CURLINE% <> MINWINDOW% THEN
        CALL FDV$PFT( FDV$_K_FT_SBK, 'NUMBER' )
    ELSE
        MINWINDOW% = MINWINDOW% - 1
        MAXWINDOW% = MAXWINDOW% - 1
        CALL FDV$PFT( FDV$_K_FT_SBK, 'NUMBER', REGARRAY$( MINWINDOW% ) )
13785 CURLINE% = CURLINE% - 1
13790 FNEND
13795 !
14000 DEF FN.VUEACT
14005 CALL FDV$PUTL( 'View account not implemented yet' )
14010 CALL FDV$WAIT
14015 FNEND
14020 !
15000 DEF FN.GETSTA
15025 CALL FDV$STAT( FMSSTAT%, RMSSTAT% )
15030 IF FMSSTAT% > 0 THEN FNEXIT
15730 CALL FDV$DTERM( TCA%() )
15735 PRINT "FDV ERROR."
15740 PRINT "", "FMS STATUS:", FMSSTAT%
15740 PRINT "", "RMS STATUS:", RMSSTAT%
15747 STOP
15750 FNEND
15999 END

```

The following listing is for the user action routine VALID1, which can exist as a separate file or in the same file as the subset application.

```

16005     FUNCTION INTEGER VALID1
16089     DECLARE INTEGER CONSTANT      &
        FOV$_K_UVAL_SUC = 1000, !Field completion success &
        FDV$_K_UVAL_FAIL = 1001 !Field completion failure
16096     FRMNAM$ = SPACE$(31)
16097     UARVAL$ = SPACE$(80)

```

```

16098          FLDNAME$ = SPACE$(31)
16099          FVALUE$ = SPACE$(1)
16140  CALL FDV$RETCX(TCA%,WKSP%,FRMNAM$,UARVAL$,CURPDS%,FLDTRM%, &
          INSOVR%,HELPNUM%
16145          CALL FDV$RETFN ( FLDNAME$, FINDEX% )
16150          CALL FDV$RET ( FVALUE$, FLDNAME$, FINDEX% )
16185          IF POS( UARVAL$, FVALUE$, 1) > 0 THEN
VALID1 = FDV$K_UVAL_SUC !Success
          ELSE
          CALL FDV$PUTL( 'Illegal value' )
          VALID1 = FDV$K_UVAL_FAIL
16210          FUNCTIONEND

```