VAX–11 FMS
Language Interface Manual

AA–N209A–TE

January 1983

This manual describes language interface issues for VAX 11 FMS (Forms Management System) application programs. VAX–11 languages discussed include BASIC, BLISS–32, C, COBOL, FORTRAN, PASCAL, and PL/I. Language-independent information is also provided for programmers writing in languages not documented in this manual.

This manual is part of the VAX–11 FMS document set that supersedes the VAX–11 FMS Version 1 document set.

Operating System: VAX/VMS Version 3.2

Software: VAX–11 FMS Version 2.0

To order additional copies of this document, contact the Software Distribution Center, Digital Equipment Corporation, Northboro, Massachusetts 01532

digital equipment corporation · maynard, massachusett

First Printing, January 1983

The information in this document is subject to change without notice and should not be construed as a commitment by Digital Equipment Corporation. Digital Equipment Corporation assumes no responsibility for any errors that may appear in this document.

The software described in this document is furnished under a license and may be used or copied only in accordance with the terms of such license.

No responsibility is assumed for the use or reliability of software on equipment that is not supplied by DIGITAL or its affiliated companies.

Printed in U.S.A.

A postage-paid READER'S COMMENTS form is included on the last page of this document. Your comments will assist us in preparing future documentation.

The following are trademarks of Digital Equipment Corporation:

	digital™	
DEC	MASSBUS	UNIBUS
DECmate	PDP	VAX
DECsystem–10	P/OS	VMS
DECSYSTEM–20	Professional	VT
DECUS	Rainbow	Work Processor
DECwriter	RSTS	
DIBOL	RSX	

M21400

Contents

Page

Preface ix

Chapter 1 Overview of the Language Interface

1.1 Form Driver Routines 1–2
1.1.1 Invoking Form Driver Routines as Procedures 1–2
1.1.2 Accessing Form Driver Status Codes as Functions 1–2
1.2 Argument Passing in FMS 1–2
1.3 Null Arguments 1–3
1.4 FMS Data Types 1–3
1.4.1 Character Strings 1–3
1.4.2 Longword Binary Integers 1–3
1.4.3 Word Binary Integers 1–4
1.5 Non-FMS Data Types 1–4
1.6 One-Dimensional Arrays 1–4
1.7 Allocation: Workspace, Terminal Control Area, and Run-Time Memory-Resident Form Area 1–4
1.8 Precautions for Using FMS 1–5
1.8.1 Memory Areas Used Exclusively by FMS 1–5
1.8.2 Why You Should Use Common Storage Areas 1–5
1.9 Data Conversion 1–6
1.10 Sample Application Program 1–6
1.10.1 Language Interface Manual 1–6
1.10.2 Other FMS Documentation 1–6

Chapter 2 Programming FMS Applications in VAX–11 BASIC

2.1 Form Driver Routines 2–2
2.1.1 Invoking Form Driver Routines as Subprograms 2–2
2.1.2 Accessing Form Driver Status Codes as Functions 2–2
2.2 Argument Passing in FMS 2–3
2.3 Null Arguments 2–3
2.4 FMS Data Types 2–3
2.4.1 Character Strings 2–3
2.4.1.1 Declaring Fixed-Length Strings 2–3
2.4.1.2 Using a Single String Variable for Multiple Forms and Fields 2–4

2.4.2 Longword Binary Integers 2–5
2.4.3 Word Binary Integers 2–5
2.5 Non-FMS Data Types 2–5
2.6 One-Dimensional Arrays 2–5
2.7 Allocation: Workspace, Terminal Control Area, and Run-Time Memory-Resident Form Area 2–6
2.8 Precautions for Using FMS 2–7
2.8.1 Memory Areas Used Exclusively by FMS 2–7
2.8.2 Precautions for Programming in Languages with Optimizing Compilers 2–7
2.9 Data Conversion 2–8
2.10 Sample Application Program in VAX–11 BASIC 2–9
2.10.1 Form Driver Definition Files 2–9
2.10.2 Command File for Building the Sample Application Program 2–10

Chapter 3 Programming FMS Applications in VAX–11 BLISS–32

3.1 Form Driver Routines 3–2
3.1.1 Invoking Form Driver Routines as Procedures 3–2
3.1.2 Accessing Form Driver Status Codes as Functions 3–2
3.2 Parameter Passing in FMS 3–2
3.3 Null Arguments 3–3
3.4 FMS Data Types 3–3
3.4.1 Character Strings 3–3
3.4.2 Longword Binary Integers 3–4
3.4.3 Word Binary Integers 3–4
3.5 Non-FMS Data Types 3–4
3.6 One-Dimensional Arrays (Vectors) 3–4
3.7 Allocation: Workspace, Terminal Control Area, and Run-Time Memory-Resident Form Area 3–5
3.8 Precautions for Using FMS 3–7
3.8.1 Memory Areas Used Exclusively by FMS 3–7
3.8.2 Why You Should Use the OWN or GLOBAL Attribute 3–7
3.8.3 Using the Form Driver as a Shareable Image 3–7
3.9 Data Conversion 3–8
3.10 Sample Application Program in VAX–11 BLISS–32 3–9
3.10.1 Form Driver Definition Files 3–9
3.10.2 Command File for Building the Sample Application Program . 3–10

Chapter 4 Programming FMS Applications in VAX–11 C

4.1 Invoking Form Driver Routines 4–2
4.2 Parameter Passing in FMS 4–2
4.3 Null Arguments 4–3
4.4 FMS Data Types 4–3
4.4.1 Character Strings 4–3
4.4.2 Longword Binary Integers 4–4
4.4.3 Word Binary Integers 4–4

4.5 Descriptors 4–4
4.5.1 Passing Arguments by Descriptor 4–5
4.5.2 String Descriptors 4–5
4.5.3 Macros 4–6
4.6 Non-FMS Data Types 4–7
4.7 One-Dimensional Arrays 4–7
4.8 Allocation: Workspace, Terminal Control Area, and Run-Time Memory-Resident Form Area 4–8
4.9 Precautions for Using FMS 4–9
4.9.1 Memory Areas Used Exclusively by FMS 4–9
4.9.2 Why You Should Use Static or External Storage Areas 4–9
4.10 Data Conversion 4–10
4.11 Sample Application Program in VAX–11 C 4–11
4.11.1 Form Driver Definition Files 4–11
4.11.2 Command File for Building the Sample Application Program . 4–12

Chapter 5 Programming FMS Applications in VAX–11 COBOL

5.1 Form Driver Routines 5–2
5.1.1 Invoking Form Driver Routines as Subroutines 5–2
5.1.2 Accessing Form Driver Status Codes as Functions 5–2
5.2 Argument Passing in FMS 5–3
5.3 Null Arguments 5–3
5.4 FMS Data Types 5–4
5.4.1 Character Strings 5–4
5.4.1.1 Passing Character Strings in FMS 5–4
5.4.1.2 String Length 5–4
5.4.2 Longword Binary Integers 5–5
5.4.3 Word Binary Integers 5–5
5.5 Non-FMS Data Types 5–5
5.6 COBOL Declarations 5–6
5.7 One-Dimensional Arrays 5–6
5.8 Allocation: Workspace, Terminal Control Area, and Run-Time Memory-Resident Form Area 5–6
5.9 Precautions for Using FMS 5–7
5.9.1 Memory Areas Used Exclusively by FMS 5–7
5.9.2 Why You Should Declare Certain Variables to Be External 5–8
5.10 Data Conversion 5–8
5.10.1 Data Conversion on PIC X Variables 5–9
5.10.2 Data Conversion on PIC 9 Variables 5–10
5.11 Sample Application Program in VAX–11 COBOL 5–10
5.11.1 Definition Files 5–10
5.11.1.1 FDVDEF.LIB 5–10
5.11.1.2 SAMPCOB.LIB 5–11
5.11.1.3 SMPCOBUAR.LIB 5–11
5.11.2 Command File for Building the Sample Application Program . 5–11

Chapter 6 Programming FMS Applications in VAX–11 FORTRAN

6.1 Form Driver Routines 6–2
6.1.1 Invoking Form Driver Routines as Subroutines 6–2
6.1.2 Accessing Form Driver Status Codes as Functions 6–2
6.2 Argument Passing in FMS 6–3
6.3 Null Arguments 6–3
6.4 FMS Data Types 6–4
6.4.1 Character Strings 6–4
6.4.2 Longword Binary Integers 6–4
6.4.3 Word Binary Integers 6–5
6.5 Non-FMS Data Types 6–5
6.6 One-Dimensional Arrays 6–5
6.7 Allocation: Workspace, Terminal Control Area, and Run-Time Memory-Resident Form Area 6–6
6.8 Precautions for Using FMS 6–7
6.8.1 Memory Areas Used Exclusively by FMS 6–7
6.8.2 Why You Should Use the COMMON Attribute 6–7
6.9 Data Conversion 6–7
6.10 Sample Application Program in VAX–11 FORTRAN 6–8
6.10.1 Form Driver Definition Files 6–8
6.10.2 Command File for Building the Sample Application Program . . 6–9

Chapter 7 Programming FMS Applications in VAX–11 PASCAL

7.1 Form Driver Routines 7–2
7.1.1 Invoking Form Driver Routines as Procedures 7–2
7.1.2 Accessing Form Driver Status Codes as Functions 7–2
7.2 Parameter Passing in FMS 7–3
7.3 Null Arguments 7–3
7.4 Entry Point Definitions 7–4
7.5 FMS Data Types 7–5
7.5.1 Character Strings 7–5
7.5.1.1 Declaring Fixed-Length Strings 7–5
7.5.2 Longword Binary Integers 7–6
7.5.3 Word Binary Integers 7–6
7.6 Non-FMS Data Types 7–6
7.7 One-Dimensional Arrays 7–6
7.8 Allocation: Workspace, Terminal Control Area, and Run-Time Memory-Resident Form Area 7–7
7.9 Precautions for Using FMS 7–8
7.9.1 Memory Areas Used Exclusively by FMS 7–8
7.9.2 Why You Should Use the VOLATILE Attribute 7–8
7.10 Data Conversion 7–8
7.11 Sample Application Program in VAX–11 PASCAL 7–9
7.11.1 Form Driver Definition Files 7–9
7.11.2 Command File for Building the Sample Application Program . 7–10

Chapter 8 Programming FMS Applications in VAX–11 PL/I

8.1 Form Driver Routines 8–2
8.1.1 Invoking Form Driver Routines as Procedures 8–2
8.1.2 Accessing Form Driver Status Codes as Functions 8–2
8.2 Argument Passing in FMS 8–3
8.3 Null Arguments 8–3
8.4 Entry Point Definitions 8–3
8.5 FMS Data Types 8–4
8.5.1 Character Strings 8–4
8.5.1.1 Defining Character Strings 8–4
8.5.2 Longword Binary Integers 8–5
8.5.3 Word Binary Integers 8–5
8.6 Declarations 8–5
8.7 Non-FMS Data Types 8–5
8.8 One-Dimensional Arrays 8–5
8.9 Allocation: Workspace, Terminal Control Area, and Run-Time Memory-Resident Form Area 8–6
8.10 Precautions for Using FMS 8–7
8.10.1 Memory Areas Used Exclusively by FMS 8–7
8.10.2 Why You Should Use the EXTERNAL Attribute 8–7
8.11 Data Conversion 8–8
8.12 Sample Application Program in VAX–11 PL/I 8–9
8.12.1 Form Driver Definition Files 8–9
8.12.2 Command File for Building the Sample Application Program . 8–10

Appendix A VAX–11 FMS Form Driver Calls

A.1 VAX–11 Language-Independent Notation A–1
A.2 Procedure Parameter Notation for Form Driver Calls A–3

Appendix B Sample Application Program Form Descriptions

Appendix C Sample Application Program Data File

Index

Preface

This manual describes the language interface between FMS and VAX–11 programming languages. The manual focuses on language-specific issues that relate to FMS application programming. Examples and precautions relating to these issues are presented. A sample application program has been written in each of the languages documented in this manual. Software for the Sample Application (SAMP) is part of the FMS Version 2 distribution kit. The code, which appears at the end of each language chapter in this manual, is included in the documentation to help you understand FMS and to help you write your programs. Examples from the Sample Application program appear throughout the text.

Intended Audience

This manual is intended for programmers who are writing programs that use FMS. FMS programs can be written in any VAX–11 programming language. We assume that programmers are experienced in the language chosen for the application program. This manual is not a tutorial in programming or a language user's guide.

Structure of This Document

This manual consists of eight chapters and three appendixes.

Chapter 1, Overview of the Language Interface, gives general information that applies to all programming languages. The chapter also describes the Sample Application program that is part of Version 2 FMS software.

Chapters 2 through 8 provide information on the language interface between FMS and seven languages. Each chapter deals with programming FMS applications in a specific language.

- Chapter 2 VAX–11 BASIC
- Chapter 3 VAX–11 BLISS–32
- Chapter 4 VAX–11 C

- Chapter 5 VAX–11 COBOL
- Chapter 6 VAX–11 FORTRAN
- Chapter 7 VAX–11 PASCAL
- Chapter 8 VAX–11 PL/I

The Sample Application in the language of each chapter and definition files for that language appear at the end of the chapters. The command file to run the Sample Application in each language is also included.

Appendix A is the table of Form Driver calls.

Appendix B provides the Sample Application form descriptions and the screen images for those forms.

Appendix C provides the data file for the Sample Application.

Using This Manual

If you program in a language documented in this manual, refer to the chapter on that language. Because each language chapter is self-contained, you can remove any chapter for easy reference. If you are writing in a language not documented in this manual, refer to Chapter 1 and Appendix A.

If you can choose a language for your application program, you might want to review the various Sample Application programs and the topics discussed for the languages that you are considering. This manual does not compare programming languages, but each chapter does describe specific programming considerations unique to that language.

Documentation Conventions

Brackets [ ]	Indicate that the item is optional.
Vertical ellipsis . . .	Indicates that not all of the statements in an example are shown.

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

Chapter 1
Overview of the Language Interface

The FMS Form Driver processes all Form Driver calls according to VAX–11 standards and language-specific rules. These rules define how arguments are passed to the Form Driver and how values are returned to your program. Your application program must comply with the requirements of the VAX–11 FMS interface. Topics discussed in each chapter include:

- Form Driver Routines
  - Invoking Form Driver Routines as Procedures
  - Accessing Form Driver Status Codes as Functions
- Argument Passing in FMS
- Null Arguments
- FMS Data Types
  - Character Strings
  - Longword Binary Integers
  - Word Binary Integers
- Non-FMS Data Types
- One-Dimensional Arrays
- Allocation: Workspace, Terminal Control Area, and Run-Time Memory-Resident Form Area
- Precautions for Using FMS
- Data Conversion
- Sample Application Program

These topics are discussed briefly in this chapter, with language-specific details given in later chapters. Programmers who want to use a VAX–11 language not documented in this manual should read this chapter for information that they will need to know.

An FMS application program can be written in any VAX–11 language with the aid of Appendix A. Appendix A contains important language interface information: the calling sequence for each Form Driver call, data access codes, data types, and passing mechanisms are presented in language-independent notation as specified by the VAX–11 Procedure Calling and Condition Handling Standard.

1.1 Form Driver Routines

All Form Driver routines are called as procedures or as functions. Syntax follows the standard requirements of your VAX–11 language.

1.1.1 Invoking Form Driver Routines as Procedures

You use a procedure call statement to invoke an FMS Form Driver routine. The call statement transfers control to an entry point of an FMS procedure, optionally passes arguments to it, and stores the location of the calling program for an eventual return. Call statements must follow the VAX–11 Procedure Calling and Condition Handling Standard. For more detail, refer to Appendix C of the *VAX–11 Run-Time Library Reference Manual.*

1.1.2 Accessing Form Driver Status Codes as Functions

An FMS status code is returned to the calling program at the completion of all Form Driver calls. The returned FMS status code can be accessed in several ways. (See the *VAX–11 FMS Form Driver Reference Manual* for a discussion of status return methods.)

Most languages have some method of making the returned value available to the program. In many languages the FMS status code is available to the calling program if you specify the routine with a function reference rather than with a call statement. In these cases, you use the standard syntax of your language for invoking a function. In general, the status is returned in register zero.

1.2 Argument Passing in FMS

The argument passing mechanism refers to the way in which data is passed to a called routine. The VAX–11 Procedure Calling Standard has three methods for passing arguments:

- By reference
- By descriptor
- By value

FMS routines, however, expect arguments to be passed only by reference and by descriptor.

By reference specifies that the storage location of the argument is passed to the routine. FMS expects integers to be passed by reference.

By descriptor specifies that the address of a descriptor data structure is passed to the routine. FMS expects character strings and arrays to be passed by descriptor.

1.3 Null Arguments

When the call syntax includes optional arguments and you do not wish to specify all of the information, you can use null arguments. Any optional argument can be omitted to simplify your program. In some programming languages, a comma is used as a placeholder for each null argument. In other languages, an address of 0 is assigned to each null argument. Optional arguments to the right of the last required argument can simply be omitted from the call.

1.4 FMS Data Types

1.4.1 Character Strings

The character string is one of the general data types used by FMS. You must be certain that your strings are initially declared to be long enough to accommodate your FMS data. Although FMS accepts both dynamic and fixed-length strings as arguments, it treats all strings as if they were fixed length. In other words, FMS does not alter the length of a dynamic string descriptor when the Form Driver returns values to the output arguments.

Two approaches are available for satisfying the fixed-length string constraints of FMS. One option is to declare your fixed-length strings to be the exact length of the FMS data to be returned. You can use the FMS/DESCRIPTION/BRIEF command to determine the length of the strings.

Alternatively, a single string can be used in different FMS calls to transfer data to or from several forms and fields. You must declare it to be at least as large as the longest field value string that will be returned to your program. You can also use the FMS/DESCRIPTION/BRIEF command to access this information. Use the FDV$RETLE call to return the length of the valid portion of the field value in the string variable. This length can then be used when referencing the data that has been entered in the field. A useful application of the FDV$RETLE call is in general purpose user action routines.

1.4.2 Longword Binary Integers

The longword binary integer is another general data type used by FMS. Numeric arguments must be longword binary integers. If you try to pass other numeric types to the Form Driver, the calls do not work properly. An exception is the FDV$DFKBD call (see the following section).

1.4.3 Word Binary Integers

The defkbd argument is a word integer array passed when the FDV$DFKBD routine is called. FMS expects a word integer array to be passed by descriptor.

1.5 Non-FMS Data Types

Data types that are not recognized by FMS can be used in your application program provided they are not passed to the Form Driver.

1.6 One-Dimensional Arrays

One-dimensional arrays are structures that can be used in FMS for the following arguments:

- tca (terminal control area)
- wksp (workspace)
- mloc (memory location)
- defkbd (define keyboard)

You must provide FMS with storage space for these arguments. You can do that by defining them to be:

- longword integer arrays or character strings for tca, wksp, and mloc
- word integer arrays for defkbd

1.7 Allocation: Workspace, Terminal Control Area, and Run-Time Memory-Resident Form Area

The FMS workspace, terminal control area, and run-time memory-resident form area variables are character strings or longword integer arrays. These variables should be placed in a common storage area of your program.

The Form Driver needs 12 bytes to associate user information about each workspace and terminal control area. The allocation for a run-time memory-resident form area must be the size of the form. The space for all of these variables is allocated by your application program. Note that FMS uses *only* 12 bytes of space for the workspace and terminal control area allocation. Any allocation greater than 12 bytes is wasted. You declare your workspace, terminal control area, and run-time memory-resident form area only once. FMS remembers their addresses after the addresses have been initially passed to the Form Driver.

For each workspace, the Form Driver also allocates an additional amount of storage space based on your estimate of the amount of memory needed to

store your largest form. If your estimate is too small, the Form Driver allocates more space automatically, but performance may be affected. An adequate estimate results in more efficient operation of the Form Driver. You can use the FMS/DIRECTORY/FULL command to find out how much space to allocate.

1.8 Precautions for Using FMS

1.8.1 Memory Areas Used Exclusively by FMS

The locations for terminal control area, workspace, and run-time memory-resident form area are used exclusively by FMS. The terminal control area and workspace are attached with the FDV$ATERM and FDV$AWKSP calls and remain allocated until the FDV$DTERM and FDV$DWKSP calls are issued or until the program ends. The run-time memory-resident form area, used in the FDV$READ call, remains allocated until the FDV$DEL call is issued or until the program ends. You never touch the terminal control areas, workspaces, or run-time memory-resident form areas in your program except to pass their addresses to the Form Driver.

1.8.2 Why You Should Use Common Storage Areas

Parameters to the following Form Driver routines should be used with caution:

FDV$ATERM	Attach terminal
FDV$AWKSP	Attach form workspace
FDV$READ	Read form into memory
FDV$SSRV	Specify status reporting variables

For example, once an FDV$SSRV call is issued, the program variables that contain the FMS status and RMS status become volatile and can change at any call point. As a general rule, you should place the status reporting variables in static storage.

In cases where you need both the FMS and RMS statuses, the FDV$STAT routine can be used. Note that only the FDV$STAT and FDV$SSRV calls provide RMS status. With the FDV$STAT routine, you do not have to worry about volatility.

The locations for terminal control area, workspace, run-time memory-resident form area, and status reporting variables must all continue to exist while the Form Driver is using them. They must remain allocated until the terminal control area and workspace are detached, until forms in memory location are deleted, and until the status reporting variables are no longer used. The variables can be protected by placing them in a common storage area; otherwise, the compiler might place them in dynamic storage.

1.9 Data Conversion

FMS uses only ASCII character strings to display data. All information displayed on the terminal screen, and all information received from the terminal operator, is represented as ASCII field values. Any manipulation of numeric data requires conversion of ASCII character strings to numeric data, and conversion of numeric data back to ASCII character strings. You can utilize your built-in language functions for conversion operations, or you can create your own conversion functions.

1.10 Sample Application Program

1.10.1 Language Interface Manual

The Sample Application program appears with each language chapter — written in that language. Additional related files are also presented.

- **Command File for Building the Sample Application Program** Includes all the information that you need to compile and link the program. The command file precedes the source listing of the Sample Application program. When FMS is installed, the command file is placed in the directory FMS$EXAMPLES.
- **VAX–11 FMS Sample Application Program** Serves as a demonstration program and is included in the FMS distribution kit. When FMS is installed, the sample program for each of the documented languages is placed in the directory FMS$EXAMPLES. The Sample Application shows most of the features provided by FMS. It is designed to serve as a learning tool. Examples from the sample program appear throughout this manual.
- **Definition Files or Other Include Files for the Sample Program** Are part of the Sample Application program package. When FMS is installed, these files are placed in the directory FMS$EXAMPLES. These files contain a variety of codes for the Form Driver routines used in the Sample Application program. Although these files have been created for use in the sample program, they can provide you with a helpful starting point as you create definitions for your own application program.

Other reference materials relating to the Sample Application program appear as appendixes. Appendix B shows the form descriptions and screen images. Appendix C shows the data file.

1.10.2 Other FMS Documentation

Examples from the Sample Application in BASIC (SAMP.BAS) appear throughout the FMS document set. The *Introduction to VAX–11 FMS* in Chapter 2 takes the reader step-by-step through the Sample Application and points out the capabilities of FMS as the program runs. Later chapters discuss the code that performs specific operations such as scrolling and using Named Data.

Chapter 2
Programming FMS Applications in VAX–11 BASIC

The FMS Form Driver processes all Form Driver calls according to VAX–11 standards and language-specific rules. These rules define how arguments are passed to the Form Driver and how values are returned to your program. Language-specific information is briefly presented in this manual. For more detail, refer to the VAX–11 BASIC document set.

Your VAX–11 BASIC application program must comply with the requirements of the VAX–11 BASIC FMS interface. Topics discussed in this chapter include:

- Form Driver Routines
  - Invoking Form Driver Routines as Subprograms
  - Accessing Form Driver Status Codes as Functions
- Argument Passing in FMS
- Null Arguments
- FMS Data Types
  - Character Strings
  - Longword Binary Integers
  - Word Binary Integers
- Non-FMS Data Types
- One-Dimensional Arrays
- Allocation: Workspace, Terminal Control Area, Run-Time Memory-Resident Form Area
- Precautions for Using FMS
- Data Conversion
- Sample Application Program in VAX–11 BASIC

A sample program written in BASIC (SAMP.BAS) appears at the end of this chapter. Following the code for SAMP.BAS are Form Driver definition files created for SAMP.BAS. Command file information needed to build the Sample Application program is in Section 2.10.2.

Examples from the Sample Application are used throughout the text to illustrate language issues. Where appropriate examples from SAMP.BAS do not exist, other examples are provided.

2.1 Form Driver Routines

You can call any FMS routine as a subprogram or as a function. Syntax follows standard VAX–11 BASIC requirements.

2.1.1 Invoking Form Driver Routines as Subprograms

You use the procedure call statement to invoke an FMS Form Driver routine. For example:

```
12235  CALL FDV$WAIT
```

Calls the Form Driver routine FDV$WAIT and passes no arguments.

```
5070  CALL FDV$GET (OPTION$, TERMINATOR%, 'OPTION')
```

Calls the Form Driver routine FDV$GET and passes three arguments.

See Appendix A for a complete list of Form Driver calls. The calling sequence for each Form Driver routine, data access codes, data types, and passing mechanisms are presented in language-independent notation as specified by the VAX–11 Procedure Calling and Condition Handling Standard. For further detail about the VAX–11 Procedure Calling and Condition Handling Standard, refer to the *VAX–11 Run-Time Library Reference Manual.*

2.1.2 Accessing Form Driver Status Codes as Functions

An FMS status code is returned to the calling program at the completion of all Form Driver calls. To receive the returned status code from a Form Driver routine, you activate the routine with a function reference rather than with a call statement. Note that this returns a standard VMS status code. For portability, other status mechanisms can also be used. (For more information, see the *VAX–11 FMS Form Driver Reference Manual*, Chapter 2.)

You declare an FMS function in an EXTERNAL LONG FUNCTION statement. The following statements declare and call FDV$GET as an FMS function:

```
EXTERNAL LONG FUNCTION FDV$GET
RETURN_STATUS = FDV$GET (OPTION$, TERMINATOR%, 'OPTION')
```

2.2 Argument Passing in FMS

The argument passing mechanism refers to the way in which data is passed to a called routine. The VAX–11 Procedure Calling Standard has three methods for passing arguments:

- By reference
- By descriptor
- By value

FMS routines, however, expect arguments to be passed only by reference and by descriptor.

By reference specifies that the storage location of the argument is passed to the routine. FMS expects integers to be passed by reference, which is the BASIC default passing mechanism for integers.

By descriptor specifies that the address of a descriptor data structure is passed to the called routine. FMS expects character strings and arrays to be passed by descriptor, which is the BASIC default passing mechanism for character strings and arrays.

2.3 Null Arguments

When the call syntax includes optional arguments and you do not wish to specify all of the information, you can use null arguments. Any optional argument can be omitted to simplify your program. A comma functions as a placeholder for each null argument. Optional arguments to the right of the last required argument can simply be omitted from the call. In the following example, the FDV$GETAL call passes only the field terminator value:

```
14058  CALL FDV$GETAL ( , TERMINATOR%)
```

2.4 FMS Data Types

2.4.1 Character Strings

The character string is one of the general data types used by FMS. For example, the FDV$GET call passes the character strings for field value (OPTION$) and field name ('OPTION'):

```
5070  CALL FDV$GET (OPTION$, TERMINATOR%, 'OPTION')
```

2.4.1.1 Declaring Fixed-Length Strings — You must be certain that your strings are initially declared to be long enough to accommodate your FMS data. Although FMS accepts both dynamic and fixed-length strings as arguments, it treats all strings as if they were fixed length. In other words, FMS does not alter the length of a dynamic string descriptor when the Form Driver returns values to the output arguments.

The initial length of a dynamic string is zero. Thus, if the Form Driver is passed a dynamic string that has not had any value assigned to it, the string being passed is a string of length zero. But the Form Driver has nonzero length data to return to you. The data cannot fit. One solution to this problem is to pre-extend any dynamic string to the exact length of the FMS data to be returned. You can use the FMS/DESCRIPTION/BRIEF command to determine the length of the strings. Once a dynamic string has been assigned a nonzero length, it can be used by FMS.

Pre-extension of dynamic strings is done throughout the Sample Application program. Many strings are pre-extended using blank spaces that correspond to the string length desired. In the following example, the strings FIRSTL$ and LASTL$ are pre-extended to length 3 by enclosing 3 spaces in apostrophes.

```
11937  FIRSTL$ = '   '
11938  LASTL$ = '   '
11940  CALL FDV$RETDN ('FIRST', FIRSTL$)
11945  CALL FDV$RETDN ('LAST', LASTL$)
```

You can also pre-extend a string using the BASIC function SPACE$. SPACE$ returns a string containing a specified number of spaces. In the following example, the SPACE$ function pre-extends the string PASS WORD$ to 12 spaces:

```
14060  PASSWORD$ = SPACE$(12)
14062  CALL FDV$RET (PASSWORD$, 'SECRET')
```

As an alternative to the above procedures, you may choose to declare fixed-length strings with the MAP and COMMON statements. Because these statements are used to allocate static storage, any strings specified are of fixed length. Thus, all strings in the following example are fixed-length strings:

```
215  MAP (ACCOUNT) ACCOUNT$ = 151
220  MAP (ACCOUNT) ACCTNO$ = 5, ACCDATE$ = 7, LAST$ = 20,     &
                   FIRST$ = 15, MIDDLE$ = 15, STREET$ = 30,D &
                   CITY$ = 20, STATE$ = 2, ZIP$ = 5,         &
                   HOMEPH$ = 10, WORKPH$ = 10, OPW$ = 12
222  MAP (TACCOUNT) TEMPACCOUNT$ = 151
```

2.4.1.2 Using a Single String Variable for Multiple Forms and Fields — There is another approach you can use to satisfy FMS fixed-length requirements. A single string variable can be used in different FMS calls to transfer data to or from several forms and fields. You must declare the string variable to be at least as large as the longest field value string that will be returned to your program. You can use the FMS/DESCRIPTION/BRIEF command to get this information. Use the FDV$RETLE call to return the length of the valid portion of the field value in the string variable. This length can then be used when referencing the data that was entered in the field.

```
100  MAP (ACCOUNT) ACCOUNT$ = 100
 .
 .
 .
400  CALL FDV$GET (ACCOUNT$, TERMINATOR%, 'FIELD')
```

```
410  CALL FDV$RETLE (LENGTHFIELD%, 'FIELD')
 .
 .
 .
500  FLDVALUE$ = SEG$ (ACCOUNT$, 1%, LENGTHFIELD%)
```

After the execution of the FDV$RETLE call, LENGTHFIELD% is equal to the length of the field named 'FIELD'. It is also equal to the valid portion of the variable ACCOUNT$. LENGTHFIELD% can now be used when referencing the data that was entered in the field named 'FIELD', and that is now in the variable ACCOUNT$. If you do not use the BASIC SEG$ function when referencing ACCOUNT$, you will reference the entire variable, including any blanks used by the Form Driver to pad the string.

A useful application of the FDV$RETLE call is in general purpose user action routines.

2.4.2 Longword Binary Integers

The longword binary integer is another general data type used by FMS. For example, the FDV$ATERM call passes the longword value for terminal control area size (12) and logical I/O channel number (2):

```
1040  CALL FDV$ATERM (TCA%(), 12%, 2%)
```

Numeric arguments must be longword binary integers. If you try to pass other numeric types to the Form Driver, the calls do not work properly. An exception is the FDV$DFKBD call (see the following section).

2.4.3 Word Binary Integers

The defkbd argument is a word integer array passed when the FDV$DFKBD routine is called. FMS expects arrays to be passed by descriptor.

2.5 Non-FMS Data Types

BASIC data types that are not recognized by FMS can be used in your BASIC application program provided they are not passed to the Form Driver.

2.6 One-Dimensional Arrays

One-dimensional arrays are structures that can be used in FMS for the following arguments:

- tca (terminal control area)
- wksp (workspace)
- mloc (memory location)
- defkbd (define keyboard)

You must provide FMS with storage space for these arguments. You can do that by defining them to be:

- longword integer arrays or character strings for tca, wksp, and mloc
- word integer arrays for defkbd

In the Sample Application program, the tca, wksp, and mloc arguments are passed to several Form Driver routines. These arguments are defined as integer array variables. You may alternatively define these variables to be character strings. (The strings can be static or dynamic but must be extended to the proper length.)

The following declarations establish names and storage for the integer array variables WORKSPACE%, CHECKWKSP%, TCA%, and MENU_FORM%:

```
130  DIM WORKSPACE% (3)       !General workspace
135  DIM CHECKWKSP% (3)       !Check workspace
140  DIM TCA% (3)             !Terminal Control Area
145  DIM MENU_FORM% (500)     !Storage for memory-resident form
```

Note that in BASIC, when an entire array is referenced, the array name must be followed by a set of parentheses () to distinguish the array from a scalar of the same name. Thus, when FDV$ATERM passes the entire array TCA%, the array name is written as follows:

```
1040  CALL FDV$ATERM (TCA%(), 12%, 2%)
```

2.7 Allocation: Workspace, Terminal Control Area, and Run-Time Memory-Resident Form Area

The FMS workspace, terminal control area, and run-time memory-resident form area variables are character strings or longword integer arrays. These variables should be placed in COMMON. Note that this is not done in the Sample Application program. The sample program's structure protects the workspaces, terminal control areas, and run-time memory-resident form areas implicitly.

The Form Driver needs 12 bytes to associate user information about each workspace and terminal control area. The allocation for a run-time memory-resident form area must be the size of the form. The space for all of these variables is allocated by your application program. Note that FMS uses *only* 12 bytes of space for the workspace and terminal control area allocation. Any allocation greater than 12 bytes is wasted. You declare your workspace, terminal control area, and run-time memory-resident form area only once. FMS remembers their addresses after the addresses have been initially passed to the Form Driver.

For each workspace, the Form Driver also allocates an additional amount of storage space based on your estimate of the amount of memory needed to

store your largest form. If your estimate is too small, the Form Driver allocates more space automatically, but performance may be affected. An adequate estimate results in more efficient operation of the Form Driver. You can use the FMS/DIRECTORY/FULL command to find out how much space to allocate.

In the following example from the Sample Application program, the workspace is allocated and the FDV$AWKSP routine is called. In the DIM statement, 12 bytes (3 longwords) are allocated to workspace. When the FDV$AWKSP routine is called, the first argument (WORKSPACE%) specifies the area of memory to be used for your workspace. The second argument specifies an estimate of the workspace size (2000 bytes) that you will need to display the largest form in your application.

```
130  DIM WORKSPACE% (3)              ! General workspace
135  DIM CHECKWKSP% (3)              ! Check workspace

1042 CALL FDV$AWKSP (CHECKWKSP%(), 2000%)
1045 CALL FDV$AWKSP (WORKSPACE%(), 2000%)
```

2.8 Precautions for Using FMS

2.8.1 Memory Areas Used Exclusively by FMS

The locations for terminal control area, workspace, and run-time memory-resident form area are used exclusively by FMS. The terminal control area and workspace are attached with the FDV$ATERM and FDV$AWKSP calls and remain allocated until the FDV$DTERM and FDV$DWKSP calls are issued or until the program ends. The run-time memory-resident form area, used in the FDV$READ call, remains allocated until the FDV$DEL call is issued or until the program ends. You never touch the terminal control areas, workspaces, or run-time memory-resident form areas in your program except to pass their addresses to the Form Driver.

2.8.2 Precautions for Programming in Languages with Optimizing Compilers

Because VAX–11 BASIC is not an optimizing compiler, your programs in VAX–11 BASIC will work without your taking the following precautions. If, however, the compiler becomes an optimizing compiler, your program could produce incorrect results.

Parameters to the following Form Driver routines should be used with caution:

FDV$ATERM	Attach terminal
FDV$AWKSP	Attach form workspace
FDV$READ	Read form into memory
FDV$SSRV	Specify status reporting variables

For example, once an FDV$SSRV call is issued, the program variables that contain the FMS status and RMS status become volatile and can change at any call point. As a general rule, you should place the status reporting variables in static storage.

In cases where you need both the FMS and RMS statuses, the FDV$STAT routine can be used. Note that only the FDV$STAT and FDV$SSRV calls provide RMS status. With the FDV$STAT routine, you do not have to worry about volatility.

The locations for terminal control area, workspace, run-time memory-resident form area, and status reporting variables must all continue to exist while the Form Driver is using them. They must remain allocated until the terminal control area and workspace are detached, until forms in memory location are deleted, and until the status reporting variables are no longer used. The variables can be protected by placing them in a common storage area; otherwise, the compiler might place them in dynamic storage.

2.9 Data Conversion

FMS uses only ASCII character strings to display data. All information displayed on the terminal screen, and all information received from the terminal operator, is represented as ASCII field values. Any manipulation of numeric data requires conversion of ASCII character strings to numeric data, and conversion of numeric data back to ASCII character strings. In the following discussion of conversion routines, you should assume that the receiving data type can support the largest number that is likely to be generated.

In the Sample Application, the following steps are taken to get a new account balance after writing a check:

```
11390   CALL FDV$RET (RI.AMTPAY$, 'AMTPAY')
11395   AMTPAY% = VAL (RI.AMTPAY$)
11400   BALANCE% = BALANCE% - AMTPAY%
11405   TOTPAY% = TOTPAY% + AMTPAY%
11410   CALL FDV$PUT (FN.CENTS$ (BALANCE%), 'BALANCE')

15900   DEF FN.CENTS$ (CENTS%)
15950   CENTS$ = FORMAT$ (CENTS%, "#######")
15955   FN.CENTS$ = XLATE (CENTS$, STRING$ (32%, 0%) + '0' + &
                  STRING$ (15%, 0%) + '0123456789')
15960   FNEND
```

In this example, the BASIC VAL function is used to convert the string expression RI.AMTPAY$ to an integer variable AMTPAY%, which is used to hold the data item's value. The integer value of the variable AMTPAY% is subtracted from the integer value BALANCE% to produce a new value for BALANCE%. The value of AMTPAY% is also added to the integer value of the variable TOTPAY% to produce a new value for TOTPAY%.

After the data operations have been completed, SAMP.BAS calls the user-created function FN.CENTS$. In that function, the BASIC FORMAT$

function converts a number to a character string with leading blanks. The BASIC XLATE function is then used to replace all blanks with zeros.

The value for the balance is displayed in the right-justified field 'BALANCE'. The rightmost digit from the program is displayed in the field's rightmost character position. The remaining digits of the character expression are placed to the left of the rightmost digit. If output is longer than the field, FMS truncates on the left. (The Form Driver displays a data length error message (FDV$_DLN) only if you have set FMS Debug mode.)

Note that in this example the output goes to a field with a decimal point field-marker character. In the presence of a decimal point field marker, the Form Driver creates strange-looking output for single-digit data items. The output will be a period followed by a space and then the digit — rather than .01, for example. In the above example, the FORMAT$ and the XLATE functions are used to prevent this kind of unconventional output.

The BASIC built-in function STR$ converts a number to a character string with no leading blanks. Your program can use the STR$ function for data conversion operations if field markers will not create a confusing appearance.

For other conversion options, see the general conversion routines in the *VAX–11 Run-Time Library Reference Manual.*

2.10 Sample Application Program in VAX–11 BASIC

The FMS Sample Application program (SAMP.BAS) is part of the FMS distribution kit. When FMS is installed, SAMP.BAS is placed in the directory FMS$EXAMPLES. Designed to be a demonstration program and learning tool, SAMP.BAS shows most of the features provided by FMS. The entire Sample Application program appears at the end of this chapter.

2.10.1 Form Driver Definition Files

The file FDVDEF.BAS is part of the Sample Application program package. When FMS is installed, FDVDEF.BAS is placed in the directory FMS$EXAMPLES. The FDVDEF.BAS file appears after the Sample Application source code.

FDVDEF.BAS contains a variety of codes for the Form Driver routines used in the Sample Application program. Although these codes have been created for use in SAMP.BAS, they can provide you with a helpful starting point as you create definitions for your own application program. The file FDVDEF.BAS includes:

- FMS terminator codes
- Function key terminators returned from the FDV$GET and FDV$WAIT calls
- Form Driver key functions for use with the FDV$DFKBD call

- User action routine (UAR) return codes, which are returned by the UARs to the Form Driver:

  Field completion UAR return codes

  Help UAR return codes

  Function key UAR return codes

- VMS status codes returned when Form Driver routines are called as functions. These codes can be signaled.
- FMS status codes returned when the FDV$STAT routine is called as a function
- Declarations of Form Driver routines

2.10.2 Command File for Building the Sample Application Program

The command file for building the Sample Application program includes all the information that you need to compile and link SAMP.BAS. When FMS is installed, the command file is placed in the directory FMS$EXAMPLES.

```
$!       S A M P B A S . C O M
$!
$!       Compile and link the BASIC version of the FMS V2 Sample Application
$!
$!       The BASIC source files are:      SAMP.BAS
$!
$!       SMPVECTOR.OBJ and SMPMEMRES.OBJ were produced by the FMS commands:
$!
$!         $ FMS/VECTOR/OUTPUT=SMPVECTOR SAMP.FLB
$!         $ FMS/MEMORY/OUTPUT=SMPMEMRES SAMP.FLB/FORM=(HELP_KEYS,HELP_MENU)
$!
$ BASIC SAMP
$ LINK  SAMP, FMS$EXAMPLES:SMPVECTOR, FMS$EXAMPLES:SMPMEMRES
```

```
1       !++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
2       ! SAMP -- The FMS V2 Sample Application Program
3       !++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
110     ! Data definitions
115     !
120     ! FMS related
125     !-
130     DIM WORKSPACE%( 3 )                 !General workspace
135     DIM CHECKWKSP%( 3 )                 !Check workspace
140     DIM TCA%( 3 )                       !Terminal Control Area
145     DIM MENU_FORM%( 500 )               !Storage for memory resident form
150     DIM CHECK_FORM%( 750 )              !Storage for memory resident form
155     DIM DPOSIT_FORM%( 500 )             !Storage for memory resident form
200     !-
205     ! Account (Read in from file)
210     !+
215     MAP( ACCOUNT ) ACCOUNT$ = 151
220     MAP( ACCOUNT ) ACCTNO$ = 5, ACCDATE$ = 7, LAST$ = 20, FIRST$ = 15, &
                       MIDDLE$ = 15, STREET$ = 30, CITY$ = 20, STATE$ = 2, &
                       ZIP$ = 5, HOMEPH$ = 10, WORKPH$ = 10, OPW$ = 12
222     MAP( TACCOUNT ) TEMPACCOUNT$ = 151
230     !+
235     ! Deposit data (Read via FDV$GETAL)
240     !-
245     MAP( DEPOSIT )  DEPOSIT$ = 80
250     MAP( DEPOSIT )  DEP.DATE$ =    7,          &
                        DEP.CURBAL$ =  6,          &
                        DEP.AMT$ =     6,          &
                        DEP.NEWBAL$ =  6,          &
                        DEP.MEMO$ =    35
260     !+
265     ! Money.
270     ! Note that all money is kept internally as integers (in cents).
275     ! It is only when the quantities are output that they look like
280     ! dollars, since all the money fields have periods as field
285     ! markers in the right places and they are right justified or
287     ! fixed decimal.
290     !
305     ! Register data.
310     ! It would be most convenient to be able to define an array
315     ! of structures for the register, but it can't be done
320     ! in BASIC (it can be done for some other languages).  What's one
```

```
325     ! instead is to define a single structure into which to put data via
330     ! structure names and also an array of strings.  After data has been
335     ! be put into the structure, it is copied to the array for convenience
340     ! in scrolling.
345     !-
350     MAP ( REGISTERITEM ) REGITEM$ =    64
355     MAP ( REGISTERITEM ) RI.NUM$ =      4,          &
                             RI.DATE$ =     7,          &
                             RI.MEMPAYTO$ =35,          &
                             RI.AMTDEP$ =   6,          &
                             RI.AMTPAY$ =   6,          &
                             RI.BALANCE$ =  6
380     DECLARE INTEGER CONSTANT REGSIZE = 30
385     DIM REGARRAY$( 30 )
500     !+
505     ! Other variables
510     !       TERMINATOR%     Terminator returned by FDV
515     !       BALANCE%        Balance in account, numeric
522     !       SBALANCE%       Starting balance
523     !       AMTPAY%         Check payment amount
525     !       TOTDEP%         Total deposits made in this session
530     !       TOTPAY%         Total checks payed in this session
535     !       OPTION$         Choice returned from menu
540     !       FMSSTATUS%      Status for last FDV call
545     !       RMSSTATUS%      RMS Status for last FDV call
550     !       LASTREGNUM%     Last number used in the register (1...REGSIZE)
555     !       LASTCHNUM%      Last check number used
560     !       FIRSTL$         First line on the form of the check image
561     !                       (from named data)
565     !       LASTL$          Last line on the form of the check image
566     !                       (from named data)
570     !       LINE$           Line return as image of form for check print
575     !       I%              Index into lines of check
580     !       NSCROL$         Number lines in scrolled area (from named data)
582     !       NSCROL%         "
585     !       CURLINE%        Line of check register that cursor is now on
590     !       MINWINDOW%      Smallest line of register being displayed
591     !                       on the scrolled area
```

```
595     !       MAXWINDOW%      Largest line of register beig displayed
596     !                       on the scrolled area
600     !       FAKE$           Value returned from fake field in scrolled area
610     !       PASSWORD$       Password from account
615     !       JUNK$           Temporary storage for return from GETAL
620     !       INSOVR%         Insert/overstrike mode
625     !       DONE$           Form done message for Deposit
627     !       FIELDNAME$      Name of field from FDV$GETAF
630     !-
900     !+
905     ! FMS terminator codes
915     !-
920     DECLARE INTEGER CONSTANT                                        &
                        FDV$K_FT_NTR    = 0,        !RETURN or ENTER    &
                        FDV$K_FT_SNX    = 6,        !Tab out of scroll line &
                        FDV$K_FT_SPR    = 7,        !Bcksp out of scr line  &
                        FDV$K_FT_SFW    = 8,        !Down arrow (scroll fwd)&
                        FDV$K_FT_SBK    = 9,        !Up arrow (scroll back) &
                        FDV$K_KP_PER    = 110,      !Keypad period      &
                        FDV$K_KP_0      = 112       !Keypad zero
```

```
995     !*****************************************************************************
996     !   SAMP main routine
997     !*****************************************************************************
1000    !+
1005    ! Initialize FMS
1010    !   Attach default terminal
1015    !   Attach normal and check workspaces (order important for help
1017    !       and refresh during CHECK/CHKDON time--try switching and see).
1020    !   Open form library, attach to channel 1
1025    !   Set keypad mode to application
1027    !   Set signal mode to bell (default, but it's fun to do)
1030    !-
1040    CALL FDV$ATERM( TCA%(), 12, 2% )                    \ C=FN.GETSTA
1042    CALL FDV$AWKSP( CHECKWKSP%(), 2000% )               \ C=FN.GETSTA
1045    CALL FDV$AWKSP( WORKSPACE%(), 2000% )               \ C=FN.GETSTA
1050    CALL FDV$LOPEN( 'FMS$EXAMPLES:SAMP', 1% )           \ C=FN.GETSTA
1055    CALL FDV$SPADA( 1% )
1060    CALL FDV$SSIGQ( 0% )
1072    !+
1074    ! Set all future calls to return status to the two status recording
1076    ! variables FMSSTATUS% and RMSSTATUS% without having to call the
1078    ! the FDV$STAT routine.
1080    !-
1082    CALL FDV$SSRV( FMSSTATUS%, RMSSTATUS% )
1086    !+
1088    ! Read in a few forms from the form library onto the dynamic
1090    ! resident form list.  You may be able to detect the difference
1091    ! in the form to form access times for those forms which have to be
1092    ! accessed from the form library on disk and those forms which are
1093    ! on the dynamic or static memory resident form list.  See the
1094    ! installation notes for this program (the LINK command) to see
1095    ! which forms are on the static memory resident form list.
1096    !-
1097    CALL FDV$READ( 'MENU', MENU_FORM%(), 2000%, SIZE_MENU% )
1098    CALL FDV$READ( 'CHECK', CHECK_FORM%(), 3000%, SIZE_CHECK% )
1099    CALL FDV$READ( 'DEPOSIT', DPOSIT_FORM%(), 2000%, SIZE_CHECK% )
1101    !+
1105    ! Initialize account information
1110    !-
1115    C=FN.INACCT
1125    !+
1130    ! Put up welcome form, wait for response
```

```
1135    !-
1140    CALL FDV$CDISP( 'WELCOME' )              \ C=FN.SRVCHK
1141    CALL FDV$WAIT
1155    !+
1160    ! Process all menu requests
1165    !-
1170    C=FN.MENU
1180    !+
1185    ! Clean up and leave:
1186    !   Close form library.
1187    !   Reset Keypad to numeric.
1188    !   Delete a form from dynamic mem. res. form list just to show how.
1189    !   Detach workspaces (not really necessary since DTERM would do it).
1190    !   Detach terminal.
1195    !-
1200    CALL FDV$LCLOS
1205    CALL FDV$SPADA( 0% )
1207    CALL FDV$DEL( 'MENU' )
1208    CALL FDV$DWKSP( WORKSPACE%() )
1212    CALL FDV$DWKSP( CHECKWKSP%() )
1215    CALL FDV$DTERM( TCA%() )
1220    GOTO 15999
```

```
4000    DEF FN.INACCT
4001    !+*********************************************************************
4010    ! Suproutine INACCT
4015    !       Read from file SAMP.DAT into internal variables.
4020    !       Set up the workspace for checks and fill in the check form
4025    !       with the account's name, address, and account number.
4030    !-*********************************************************************
4034
4035    !+
4040    ! Open file, set account data
4045    !-
4047    ON ERROR GOTO 4500
4050    OPEN 'FMS$EXAMPLES:SAMP.DAT' FOR INPUT AS FILE #5%, ACCESS READ
4055    LINPUT #5%, ACCOUNT$
4065    !+
4070    ! Read the remaining records into the check register, counting them.
4075    ! The last record has the current balance, and some record has the
4080    ! last check number used (not necessarily the last record).
4085    ! Note that in BASIC the record is read into the array and reference
4090    ! to the check number is via a substring rather than symbolically.
4091    ! Other languages may access differently.
4095    !-
4100    LASTCHNUM% = 0
4105    LASTREGNUM% = 0
4107    WHILE LASTREGNUM% < REGSIZE
4110        LINPUT #5%, REGARRAY$( LASTREGNUM% + 1 )
4112        LASTREGNUM% = LASTREGNUM% + 1
4115        IF SEG$( REGARRAY$( LASTREGNUM% ), 1, 4 ) <> SPACE$(4) THEN
                LASTCHNUM% = VAL( SEG$( REGARRAY$( LASTREGNUM% ), 1, 4 ) )
4120    NEXT
4130    !+
4135    ! Reached here without hitting end of file, should probably print
4140    ! message or something, except that this is just a demo.
4145    ! As it is, just fall through and ignore remaining records.
4150    !-
4180
4200    !-
4210    ! Reach here as result of end of file--last record tried didn't read.
4211    ! Check for data file in error.
4212    ! Take balance from last record read.
4213    ! Set session sums to zero to say no activity yet.
4215    !-
```

```
4220    IF LASTREGNUM% = 0 THEN
            PRINT "DATA FILE IN ERROR"
            STOP
4225    BALANCE% = VAL( SEG$( REGARRAY$( LASTREGNUM% ), 59, 64 ) )
4230    SBALANCE% = BALANCE%
4235    TOTDEP% = 0
4240    TOTPAY% = 0
4245    !+
4250    ! Set up the check workspace once so we don't have to do it every time.
4255    !-
4260    C=FN.FMTCHK
4265
4270    FNEXIT
4495
4500    !+
4505    ! Error handler for BASIC I/O
4510    ! If it's end of file, just close file and resume at 4200.
4520    ! Otherwise, revert to BASIC error handling
4530    !-
4535    IF ERR = 11% THEN
            CLOSE #5%
            RESUME 4200
        ELSE
            ON ERROR GOTO 0
4550    FNEND


4600    DEF FN.FMTCHK
4601    !+*********************************************************************
4605    ! Subroutine FMTCHK
4610    !       Format account data onto check form in the check workspace.
4615    !-*********************************************************************
4645    CALL FDV$SWKSP( CHECKWKSP%() )
4650    CALL FDV$LOAD( 'CHECK' )
4655    CALL FDV$PUT( TRM$( FIRST$ ) + SP + SEG$( MIDDLE$, 1, 1 ) + '.' + &
                      SP + LAST$ , 'NAME' )
4660    CALL FDV$PUT( STREET$, 'STREET' )
4665    CALL FDV$PUT( TRM$( CITY$ ) + ', ' + STATE$ + SP + ZIP$ , 'CSZ' )
4670    CALL FDV$PUT( HOMEPH$, 'HOMEPH' )
4675    CALL FDV$PUT( ACCTNO$, 'ACCTNO' )
4685    CALL FDV$SWKSP( WORKSPACE%() )
4695    FNEND
```

```
5000    DEF FN.MENU
5001    !+*********************************************************************
5005    ! Subroutine MENU
5010    !       Accept inputs from the menu form and dispatch to the
5015    !       appropriate routine.  Repeat until option 1 (exit) is
5020    !       chosen.  The UARs in the form guarantee that we get back
5023    !       only inputs '1'-'5' with the correct terminators.
5024    !       Options are:
5025    !               1 => Exit
5026    !               2 => Write checks
5027    !               3 => Make deposit
5028    !               4 => View register
5029    !               5 => View account data
5030    !-*********************************************************************
5035
5040    OPTION$ = ' '   ! Extend string variable before FMS call (BASIC only)
5045    WHILE 1 = 1
5050        CALL FDV$CDISP( 'MENU' )                                \ C=FN.SRVCHK
5070        CALL FDV$GET( OPTION$, TERMINATOR%, 'OPTION' )
5075        ON VAL( OPTION$ ) 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.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    FNEND
```

```
11000   DEF FN.WRITCH
11001   !+*********************************************************************
11005   ! Subroutine WRITCH
11010   !       Write one or more checks
11015   !-*********************************************************************
11017
11018   !+
11019   ! Turn on LED 3 on the VT100 during this routine, just to show how.
11020   !-
11021   CALL FDV$LEDON( 3% )
11022
11025   !+
11027   ! Mark WORKSPACE not displayed so it doesn't show up during a refresh.
11030   ! Put up CHECK form from already loaded workspace
11032   ! and display current balance
11035   !-
11037   CALL FDV$NDISP
11040   CALL FDV$SWKSP( CHECKWKSP%() )
11045   CALL FDV$DISPW
11050
11070   CALL FDV$PUT( FN.CENTS$( BALANCE% ), 'BALANCE' )
11075   !+
11080   ! Process checks until a keypad period is read
11085   !-
11090   TERMINATOR% = 0
11095   UNTIL TERMINATOR% = FDV$K_KP_PER
11100       C=FN.ONECHK                         ! Process one check
11105       C=FN.ENDCHK                         ! Give options for continuing
11110   NEXT
11120   !+
11125   ! Turn off LED 3 on VT100
11130   !-
11135   CALL FDV$LEDOF( 3% )
11140   CALL FDV$SWKSP( WORKSPACE%() )
11150
11155   FNEND
```

```
11300   DEF FN.ONECHK
11301   !+*********************************************************************
11305   ! Subroutine ONECHK -- Process one check
11310   !       If input is terminated by Kpd Period, return with no action
11315   !       Else deduct from balance and enter into register.
11317   !       Note that a UAR in the form guarantees that the amount of
11318   !       the check is always less than or equal to the balance.
11319   !       Note that the form function key UAR  allows only Kpd Period
11320   !       as terminator (other than FDV$K_FT_NTR).
11321   !-*********************************************************************
11322   CALL FDV$PUT( STR$( LASTCHNUM% + 1 ), 'NUMBER' )
11325   CALL FDV$GETAL( JUNK$, TERMINATOR% )
11330   IF TERMINATOR% = FDV$K_KP_PER THEN FNEXIT
11332
11335   !+
11340   ! If the check wouldn't fit in the register, don't process, just
11345   ! give error message, wait for acknowledgement, and return
11350   !-
11355   IF _LASTREGNUM% = REGSIZE THEN
            CALL FDV$PUTL( "Register full, can't enter check" )
            CALL FDV$WAIT
            FNEXIT
11360
11365   !+
11370   ! Get amount from check.
11372   ! Update balance (in memory and on screen) and session sums.
11375   ! Transfer form values to register item.
11385   !-
11390   CALL FDV$RET( RI.AMTPAY$, 'AMTPAY' )
11395   AMTPAY% = VAL( RI.AMTPAY$)
11400   BALANCE% = BALANCE% - AMTPAY%
11405   TOTPAY% = TOTPAY% + AMTPAY%
11407
11410   CALL FDV$PUT( FN.CENTS$( BALANCE% ), 'BALANCE' )
11415   CALL FDV$RET( RI.BALANCE$, 'BALANCE') !Avoid need to format RI.BALANCE$
11420
11425   RI.AMTDEP$ = ''
11430   CALL FDV$RET( RI.NUM$, 'NUMBER' )
11435   CALL FDV$RET( RI.DATE$, 'DATE' )
11440   CALL FDV$RET( RI.MEMPAYTO$, 'PAYTO' )   !Note: not from check's MEMO
11445
11450   !+
```

```
11455   ! Update register array and counters
11460   ! (Note that the two step update (form->regitem->regarray)
11465   ! is necessary in BASIC, not necessarily in every language).
11470   !-
11475   LASTREGNUM% = LASTREGNUM% + 1
11480   LASTCHNUM% = LASTCHNUM% + 1
11485   REGARRAY$( LASTREGNUM% ) = REGITEM$
11490
11495   FNEND                                                          !
```

```
11600   DEF FN.ENDCHK
11601   !+*********************************************************************
11605   ! Subroutine ENDCHK
11610   !       Finish off check processing by giving operator
11615   !       three options:
11620   !               RETURN  Write another check
11625   !               KPD 0   Print the check into file SAMPCH.DAT
11630   !               KPD .   Return to menu
11631   !       Check to see if check write was aborted by Kpd per.
11632   !       If so, then don't give any further choice, just abort.
11633   !       Note that Form Function Key UAR allows only the above
11634   !       terminators to get through.
11635   !-*********************************************************************
11640   IF TERMINATOR% = FDV$K_KP_PER THEN FNEXIT
11645   !+
11650   ! Tell the operator that the check has been paid by overlaying with
11655   ! a new form, using the normal workspace, thereby saving the check
11660   ! workspace in case another check is to be written.
11665   !-
11675   CALL FDV$SWKSP( WORKSPACE%() )
11685   CALL FDV$DISP( 'CHECK_DONE' )                                  \ C=FN.SRVCHK
11700    +
11705   ! Wait for operator to enter either KPD period, NTR, or KPD zero.
11707   ! Print the check as many times as requested.
11710   ! (Note that a UAR on the form guarantees that only those terminators
11715   ! are accepted).
11720   ! Process accordingly.
```

```
11730   !-
11735   CALL FDV$WAIT( TERMINATOR% )
11740   WHILE TERMINATOR% = FDV$K_KP_0
11745       C=FN.PRICHK             Print the check
11750       CALL FDV$WAIT( TERMINATOR% )
11755   NEXT
11765   !+
11770   ! If choice is to quit,
11771   ! then mark check wksp undisplayed so it doesn't appear during refresh,
11772   ! else mark normal workspace (occupied by CHECK_DONE form) undisplayed
11773   !   so it doesn't show during refresh and then clear its lines.
11774   !   (Clearing the space occupied by the CHECK_DONE form, lines 20-23
11775   !   is better done by overlaying with a blank form to
11776   !   avoid having to know the line numbers to clear).
11780   !-
11785   IF TERMINATOR% = FDV$K_KP_PER THEN
           CALL FDV$SWKSP( CHECKWKSP%() )
           CALL FDV$NDISP
       ELSE
           CALL FDV$NDISP
           CALL FDV$CLEAR( 20%, 4% )
           CALL FDV$SWKSP( CHECKWKSP%() )
11795   !+
11800   ! Going to write another check now or eventually, so:
11815   ! Clear out operator entered fields.
11820   !-
11845   CALL FDV$PUTD( 'AMTPAY' )
11850   CALL FDV$PUTD( 'MEMO' )
11855   CALL FDV$PUTD( 'PAYTO' )
11865   FNEND
```

```
11900   DEF FN.PRICHK
11901   '+*********************************************************************
11905   ! Subroutine PRICHK
11910   !       Print the check into the file SAMPCH.DAT
11915   !       Use the check workspace, then switch back to the normal wksp
11920   !       to keep things clean.
11921    -*********************************************************************
11922   !+
11923   ! Open check writing file.  Note there's a new version for every check.
11924   ! Switch workspaces
11925   !-
11926   OPEN 'SAMPCH.DAT' FOR OUTPUT AS FILE # 2%, ACCESS WRITE, RECORDSIZE 80
11927   CALL FDV$SWKSP( CHECKWKSP%() )
11930   !+
11932   ! Get the top and bottom lines of the check from the named data
11934   ! (first two characters).
11935   ! Must pre-extend BASIC dynamic string variables before calling FMS
11936   !-
11937   FIRSTL$ = '  '
11938   LASTL$ = '  '
11940   CALL FDV$RETDN( 'FIRST', FIRSTL$ )                      \ C=FN.SRVCHK
11945   CALL FDV$RETDN( 'LAST', LASTL$ )                        \ C=FN.SRVCHK
11960   !+
11965   ! Get lines from form.
11967   ! Convert to line printer style.
11970   ! Write to file.
11975   !-
11980   FOR I% = VAL( SEG$( FIRSTL$, 1, 2 ) ) TO VAL( SEG$( LASTL$, 1, 2 ))
11981       LINE$ = SPACE$( 80 ) ! Pre-extend character variable (BASIC only)
11982       CALL FDV$RETFL( I%, LINE$, LINELENGTH% )
11986       PRINT #2%, SEG$( LINE$, 1, LINELENGTH% )
11988   NEXT I%
11990   CALL FDV$PUTL( 'Check written to file' )
11992   CLOSE #2%
11993   CALL FDV$SWKSP( WORKSPACE%() )
11995   FNEND
```

```
12000   DEF FN.MAKDEP
12001   !+*********************************************************************
12005   ! Subroutine MAKDEP
12010   !       Make a deposit, enter into check register
12015   !       Cancel on Keypad period.
12017   !       Note that the Form function key UAR allows only Kpd period.
12020   !
12035   ! Put up deposit form with current balance
12040   !-*********************************************************************
12050   CALL FDV$CDISP( 'DEPOSIT' )                         \ C=FN.SRVCHK
12065   CALL FDV$PUT( FN.CENTS$( BALANCE% ), 'CURBAL' )
12075   !+
12080   ! Get deposit amount and memo from operator.
12085   ! Abort on Kpd period.
12090   !-
12095   CALL FDV$GETAL( DEPOSIT$, TERMINATOR% )
12100   IF TERMINATOR% = FDV$K_KP_PER THEN FNEXIT
12110   !+
12115   ! Have deposit information now. If no room in check register
12120   ! must abort.
12125   !-
12130   IF LASTREGNUM% = REGSIZE THEN
            CALL FDV$PUTL( "Register full, can't enter deposit" )
            CALL FDV$WAIT
            FNEXIT
12155   !+
12160   ! Add to balance and session sum.
12161   ! Check for overflow (program and form keep only six digits).
12162   ! Display new balance.
12163   ! Make entry in register.
12165   !-
12170   BALANCE% = BALANCE% + VAL( DEP.AMT$ )
12172   TOTDEP% = TOTDEP% + VAL( DEP.AMT$ )
12174   IF BALANCE% >= 1000000 THEN
            BALANCE% = BALANCE% - 1000000
            CALL FDV$PUTL( "Overflow in bank computer, only 6 digits kept" )
            CALL FDV$WAIT
12180   CALL FDV$PUT( FN.CENTS$( BALANCE% ), 'NEWBAL' )
12185   RI.NUM$ = '                                 ! Blank since it's not a check
12190   RI.DATE$ = DEP.DATE$
12195   RI.MEMPAYTO$ = DEP.MEMO$
12197   RI.AMTDEP$ = DEP.AMT$
```

```
12200   RI.AMTPAY$ = ''
12210   CALL FDV$RET( RI.BALANCE$, 'NEWBAL' )    ! Avoids need to format RI.BALANCE$
12215   LASTREGNUM% = LASTREGNUM% + 1
12220   REGARRAY$( LASTREGNUM% ) = REGITEM$
12221   '+
12222   ! Sample of how to keep message texts stored with the form rather
12223   ! than in a program.  This is especially useful for multi-lingual
12224   ! environments: only the form text and the form named data must
12225   ! be changed and nothing in the program.  The trick is to store the
12226   ! response text in named data. This is the only example of how to do
12227   ! it in this program, but all messages could be stored like this.
12228   ! Message intent is: "Deposit made, press RETURN or ENTER to continue."
12229   !-
12230   DONE$ = SPACE$(80)       !Pre-extend string (BASIC only)
12232   CALL FDV$RETDN( 'DONE', DONE$ )
12234   CALL FDV$PUTL( DONE$ )
12235   CALL FDV$WAIT
12240   FNEND
```

```
13000   DEF FN.VUEREG
13001   '+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
13005   ! Subroutine VUEREG
13010   !       View the check register and scroll through it.
13015   !       Also display totals for current session.
13030   '
13035   ! Put up register form.
13037   ! Check for current session totals overflow. If so, output 'OVRFLO'
13040   ! Put out summary of this session into indexed(4) fields.
13045   !-----------------------------------------------------------------------
13047   CALL FDV$CDISP( 'REGISTER' )                          \ C=FN.SRVCHK
13048   IF TOTDEP% < 1000000 THEN
          DEPDSP$ = FN.CENTS$( TOTDEP% )
        ELSE
          DEPDSP$ = 'OVRFLO'
13049   IF TOTPAY% < 1000000 THEN
          PAYDSP$ = FN.CENTS$( TOTPAY% )
        ELSE
          PAYDSP$ = 'OVRFLO'
```

```
13050   CALL FDV$PUT( FN.CENTS$( SBALANCE% ), 'SUMMARY', 1% )
13055   CALL FDV$PUT( DEPDSP$,                'SUMMARY', 2% )
13060   CALL FDV$PUT( PAYDSP$,                'SUMMARY', 3% )
13065   CALL FDV$PUT( FN.CENTS$( BALANCE%  ), 'SUMMARY', 4% )
13070   !+
13075   ! Get number of lines in scroll area from form named data (item 1).
13080   !-
13085   NSCROL$ = '  '  ! Pre-extend string variable before call (BASIC only).
13090   CALL FDV$RETDI( 1%, NSCROL$ )                          \ C=FN.SRVCHK
13095   NSCROL% = VAL( NSCROL$ )
13100   !+
13105   ! Put lines from check register array into scrolled area.
13110   ! The window is initially from item 1 up to item
13115   ! min(NSCROL$,LASTREGNUM%), that is, up to the size of the scrolled
13120   ! area or the size of the register, whichever is less. Assume there
13125   ! is at least one line (the initial deposit).
13130   !-
13135   MINWINDOW% = 1
13140   CALL FDV$PUTSC( 'NUMBER', REGARRAY$(1) )          ! First line
13145   CURLINE% = 1                                      ! Res item cursor is on
13150   WHILE ( CURLINE% < LASTREGNUM% AND CURLINE% < NSCROL% )
13155       CURLINE% = CURLINE% + 1
13160       CALL FDV$PFT( FDV$K_FT_SFW, 'NUMBER' )
13165       CALL FDV$PUTSC( 'NUMBER ', REGARRAY$( CURLINE% ) )
13170   NEXT
13171   MAXWINDOW% = CURLINE%
13175   !+
13180   ! Get input from fake field of scrolled line and do what it says:
13185   !       kpd . or RETURN/ENTER   => return to menu
13190   !       UPARROW or TAB          => scroll forward
13192   !       DOWNARROW or BACKSPACE  => scroll backward
13195   !       all others              => ignore
13200   ! Note that there is no Form Function Key VAR so this routine
13205   ! handles all terminators itself (by ignoring illegal ones).
13210   !-
13215   CALL FDV$GET( FAKE$, TERMINATOR%, 'FAKE' )
13220   WHILE NOT ( TERMINATOR% = FDV$K_FT_NTR OR TERMINATOR% = FDV$K_KP_PER )
13225       IF TERMINATOR% = FDV$K_FT_SFW OR TERMINATOR% = FDV$K_FT_SNX THEN C=FN.SCRFWD
13235       IF TERMINATOR% = FDV$K_FT_SBK OR TERMINATOR% = FDV$K_FT_SPR THEN C=FN.SCRBAK
13245       CALL FDV$GET( FAKE$, TERMINATOR%, 'FAKE' )
13250   NEXT
13255   FNEND
```

```
13500   DEF FN.SCRFWD
13501   !+*********************************************************************
13505   ! Subroutine SCRFWD -- Scroll forward.
13510   !       CURLINE% is the line in the register that the cursor is on.
13512   !       MINWINDOW% and MAXWINDOW% delimit the part of the register
13513   !       currently displayed in the scrolled area
13515   !-*********************************************************************
13520
13525   !+
13530   ! If cursor is at the end of the register, report, and return
13535   !-
13540   IF CURLINE% = LASTREGNUM% THEN
            CALL FDV$PUTL( 'Last line of register' )
            FNEXIT
13545   !+
13550   ! If cursor not at the last line of a window, just move down
13555   ! If cursor is  at the last line of a window,
13560   !       move window forward one line,
13565   !       write the new last line to the last line of the scrolled area
13567   ! Move current line pointer forward
13570   !-
13580   IF CURLINE%  > MAXWINDOW% THEN
            CALL FDV$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
```

```
13698
13700   DEF FN.SCRBAK
13701   !+*********************************************************************
13705   ! Subroutine SCRBAK -- Scroll backward
13710   !       CURLINE% is the line in the register that the cursor is on.
13712   !       MINWINDOW% and MAXWINDOW% delimit the part of the register
13713   !       currently displayed in the scrolled area
13715   !-*********************************************************************
13720
13725   !+
13730   ! If the cursor is at the beginning of the register, report, and return
13735   !-
13740   IF CURLINE% = 1 THEN
            CALL FDV$PUTL( 'First line of register' )
            FNEXIT
13745   !+
13750   ! If cursor not at first line of the window, just move up
13755   ! If cursor is  at first line of the window,
13760   !       move window back one line,
13765   !       write the new first line to the first line of the scrolled area
13767   ! Move current line pointer back
13770   !-
13780   IF CURLINE% <> MINWINDOW% THEN
            CALL FDV$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
```

```
14000   DEF FN.VUEACT
14001   !+*********************************************************************
14005   ! Subroutine VUEACT
14010   !       View the account data.
14015   !       IF operator knows the secret word, let operator change
14017   !       the account data for this session.
14020   !-*********************************************************************
14025
14035   CALL FDV$CDISP( 'ACCOUNT_DATA' )                        \ C=FN.SRVCHK
14045   CALL FDV$PUTAL( ACCOUNT$ )
14046   CALL FDV$PUTD( 'SECRET')
14050   !+
14051   ! This is not the best way to do protection, just a way of showing
14052   ! another FMS feature. At this point, supervisor mode is on, so the
14053   ! only input allowed is to the password field.
14055   ! If operator doesn't know password, return to menu.
14056   !-
14058   CALL FDV$GETAL( , TERMINATOR% ) !Don't care about value now
14059   IF TERMINATOR% = FDV$K_KP_PER THEN FNEXIT
14060   PASSWORD$ = SPACE$(12)  !Pre-extend string variable (BASIC only)
14062   CALL FDV$RET( PASSWORD$, 'SECRET' )
14065   IF OPW$ <> PASSWORD$ THEN FNEXIT
14070   !+
14075   ! Allow input from other fields and read from them.
14080   ! If read is terminated by Keypad Period, don't change account.
14100   !-
14105   CALL FDV$SPOFF
14110   C=FN.SIMGTL                     ! Read all fields
14115   CALL FDV$SPON                   ! Not really needed, just showing off.
14120   IF TERMINATOR% <> FDV$K_KP_PER THEN
            CALL FDV$RETAL( ACCOUNT$ )
            C=FN.FMTCHK         ! Update the check workspace
14125   FNEND
```

```
14300   DEF FN.SIMGTL
14301    !+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
14305   ! Simulate action of FDV$GETAL, using FDV$GETAF and PFT.  Could
14310   ! replace this whole routine with a call on FDV$GETAL, but this shows
14315   ! how mainline program can allow same operator freedom of filling in
14320   ! fields but still regain control after each or changed field.
14322   ! Technique is to read any field, looking only at terminator, then do
14323   ! a process field terminator call to do the operator's action.
14324   ! This technique can be used with calls on FDV$GET or FDV$GETAF.
14325   ! This example starts with a GET on field '*', first field on form.
14326   !-----------------------------------------------------------------------
14330   CALL FDV$GET( JUNK$, TERMINATOR%,  '*' )
14335   CALL FDV$RETFN( FIELDNAME$, FIELDINDEX% )       !Get first field's name
14340   WHILE 1=1
14341       !+
14342       ! Do any special processing for field FIELDNAME$ at this point.
14343       ! ...
14344       ! Go to next or previous field or leave form
14346       !-
14350       CALL FDV$PFT( TERMINATOR% )
14357       !+
14358       ! If status is error, then PFT failed because terminator was
14359       ! a Keypad Key- which means return to caller.
14360       !-
14361       IF FMSSTATUS% < 0 THEN FNEXIT
14365       IF TERMINATOR% = FDV$K_FT_NTR THEN
                IF FMSSTATUS% <> 2 THEN
                    FNEXIT
                ELSE
                    CALL FDV$PUTL( 'INPUT REQUIRED' )
                    CALL FDV$BELL
14380       !+
14382         Go get any other field, returning its name
14384       !-
14390       CALL FDV$GETAF( JUNK$, TERMINATOR%, FIELDNAME$, FIELDINDEX% )
14400   NEXT
14410   FNEND
```

```
15000   DEF FN.GETSTA
15001   !+*********************************************************************
15005   ! Subroutine GETSTA
15010   ! Check FMS status by calling FDV$STAT.
15012   ! If not success (>0), print and stop
15015   !-*********************************************************************
15021
15025   CALL FDV$STAT( FMSSTATUS%, RMSSTATUS% )
15030   IF FMSSTATUS% <> 0 THEN FNEXIT
15035   C=FN.ERROR                          ! and never come back
15040   FNEND



15300   DEF FN.SRVCHK
15301   !+*********************************************************************
15310   ! Subroutine SRVCHK
15315   ! Check FMS status by looking at the status recording variables.
15316   !-*********************************************************************
15325   IF FMSSTATUS% > 0 THEN FNEXIT
15330   C=FN.ERROR                          ! and never come back
15335   FNEND



15700   DEF FN.ERROR
15701   !+*********************************************************************
15702   ! Subroutine ERROR
15705   ! There is an error returned in the status variables. Detach the
15710   ! terminal to clean up, then print the errors, and stop.
15715   !-*********************************************************************
15730   CALL FDV$DTERM( TCA%() )
15735   PRINT "FDV ERROR."
15740   PRINT "","FMS STATUS:",FMSSTATUS%
15745   PRINT "","RMS STATUS:",RMSSTATUS%
15747   STOP
15750   FNEND
```

```
15900   DEF FN.CENTS$( CENTS% )
15905   !+*********************************************************************
15910   ! Function FN.CENTS
15915   ! Return the string value of CENTS% suitable for outputing in a six
15920   ! wide field with two decimal places.  The important thing to note is
15925   ! that a number less than 100 should be output with leading zeros so
15930   ! that a string like "bbbbb9" doesn't display as "bbbb.b9" on the form.
15935   ! We actually convert all spaces to zero and then let the forms zero
15940   ! suppress the result.
15945   !-*********************************************************************
15950   CENTS$ = FORMAT$( CENTS%, "######" )
15955   FN.CENTS$ = XLATE( CENTS$, STRING$(32%,0%)+'0'+STRING$(15%,0%)+'0123456789' )
15960   FNEND

15999   END




16005   FUNCTION INTEGER VALID1
16010   !+*********************************************************************
16015   ! VALID1
16017   !       UAR for field validation of any one character field. The
16020   !       UAR associated data has in it the legal characters allowed,
16025   !       except that blank is not allowed unless it appears before
16030   !       the first trailing blank.  For example an assoc. value string
16035   !       'aqr' implies that only the letters a, q, and r are allowed.
16040   !       A string ' aqr' means that blank is acceptable in addition
16045   !       to a, q, and r. Note that this routine is case sensitive
16050   !       (that is, it checks for correct case).  You can get around
16055   !       case sensitivity by using the force upper case field attribute
16060   !       and putting only capitals into the UAR associated value
16065   !       string.
16070   !
16075   !       This routine can be used with any form and field since
16080   !       it determines the context for itself.
16085   !-*********************************************************************
16088   !+
16089   DECLARE INTEGER CONSTANT                                        &
          FDV$K_UVAL_SUC= 1000,   !Field completion success             &
          FDV$K_UVAL_FAI_=1001    !Field completion failure
```

```
16090   ! Pre-extend the strings into which FMS will return values
16095   !-
16096   FRMNAM$ = SPACE$(31)
16097   UARVAL$ = SPACE$(80)
16098   FLDNAME$ = SPACE$(31)
16099   FVALUE$ = SPACE$(1)
16105
16110   !+
16120   ! Retrieve context: we will ignore TCA address, WKSP address, FRMNAM$,
16125   !       CURPOS, FLDTRM, INSOVR, and HELPNUM using only UARVAL$, and
16127   !       only the initial, non-blank characters of it.
16130   ! Retrieve field name and index.
16135   ! Retrieve field value.
16140   CALL FDV$RETCX( TCA%, WKSP%, FRMNAM$, UARVAL$, CURPOS%, FLDTRM%, INSOVR%, HELPNUM% )
16142   UARVAL$ = TRM$( UARVAL$ )
16145   CALL FDV$RETFN( FLDNAME$, FINDEX% )
16150   CALL FDV$RET( FVALUE$, FLDNAME$, FINDEX% )
16160
16165   !+
16170   ! To be valid, FVALUE$ must occur in the string UARVAL$
16175   !-
16185   IF POS( UARVAL$, FVALUE$, 1) > 0 THEN
            VALID% = FDV$K_UVAL_SUC             !Success
        ELSE
            CALL FDV$PUTL( 'Illegal value' )
            VALID% = FDV$K_UVAL_FAIL
16210   FUNCTIONEND




17000   FUNCTION INTEGER TAKE%5
17010   !+*********************************************************************
17015   ! Function Key User Action Routine for the MENU form of SAMP.
17020   ! Convert Keypad 1-5 into field values 1-5.
17025   ! Convert Keypad period into field value 1.
17030   ! Reject all other function keys with error message.
17035   !-*********************************************************************
17037   DECLARE INTEGER CONSTANT                                            &
                FDV$K_KP_PER    = 110,                                      &
                FDV$K_KP_1      = 113,                                      &
```

```
                FDV$K_KP_2      = 114,                                  &
                FDV$K_KP_3      = 115,                                  &
                FDV$K_KP_4      = 116,                                  &
                FDV$K_KP_5      = 117,                                  &
                FDV$K_UKEY_ERR= 3000,    !Fn Key failure, FDV signals   &
                FDV$K_UKEY_TRM= 3001,    !FN Key success, normal f.k.   &
                FDV$K_UKEY_NTR= 3003,    !Fn Key succ, treat as ENTER   &
                FDV$K_UKEY_SUC= 3004     !Fn Key succ, ignore
17040
17045   !+
17050   ! Pre-extend the strings into which FMS will return values
17055   !-
17060   FRMNAM$ = SPACE$(4)
17062   UARVAL$ = SPACE$(1)             !No UAR value expected
17065
17070   !+
17075   ! Retrieve context: we will ignore TCA address, WKSP address, FRMNAM$,
17080   !       UARVAL$, CURPOS%, INSOVR% and HELPNUM%, using only FLDTRM%
17085   CALL FDV$RETCX( TCA%, WKSP%, FRMNAM$, UARVAL$, CURPOS%, FLDTRM%, INSOVR%, HELPNUM% )
17100
17105   !+
17110   ! Do the conversion, displaying the value converted if found.
17115   ! Reject if not one of the expected terminators.
17120   !
17130   VALUE$ = ''
17135   IF FLDTRM% = FDV$K_KP_1 THEN VALUE$ = '1'
17140   IF FLDTRM% = FDV$K_KP_2 THEN VALUE$ = '2'
17145   IF FLDTRM% = FDV$K_KP_3 THEN VALUE$ = '3'
17150   IF FLDTRM% = FDV$K_KP_4 THEN VALUE$ = '4'
17155   IF FLDTRM% = FDV$K_KP_5 THEN VALUE$ = '5'
17160   IF FLDTRM% = FDV$K_KP_PER THEN VALUE$ = '1'
17165   IF VALUE$ <> '' THEN
           CALL FDV$PUT( VALUE$, 'OPTION' )
           ! Treat as if it is RETURN
           TAKE15 = FDV$K_UKEY_NTR
        ELSE
           CALL FDV$PUTL( 'Illegal Function Key' )
           CALL FDV$SIGOP
           ! Just ignore it now
           TAKE15 = FDV$K_UKEY_SUC
17175   FUNCTIONEND
```

```
18000   FUNCTION INTEGER PASSKY
18010   !+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
18015   ! General function key var to pass only those from the (small) list
18020   ! in the var associated value string and reject all others.
18021   ! The list is of the form: n <oneblank> n <oneblank> ... n <manyblanks>
18023   ! For example the string '110 112' would accept keypad period and
18024   ! keypad zero but no other function keys.
18025   !-------------------------------------------------------------------
18030   DECLARE INTEGER CONSTANT                                          &
                FDV$K_UKEY_ERR= 3000,   !Fn Key failure, FDV signals    &
                FDV$K_UKEY_TRM= 3001,   !FN Key success, normal f.k.    &
                FDV$K_UKEY_NTR= 3003,   !Fn Key succ, treat as ENTER    &
                FDV$K_UKEY_SUC= 3004    !Fn Key succ, ignore
18045   !+
18050   ! Pre-extend the strings into which FMS will return values
18055   !-
18060   FRMNAM$ = SPACE$(4)
18062   UARVAL$ = SPACE$(82) !Two longer to ensure trailing blanks
18065
18070   !+
18075   ! Retrieve context: we will ignore TCA address, WKSP address, FRMNAM$,
18080   !       INSOVR%, HELPNUM% and CURPOS%, using only FLDTRM% and UARVAL$.
18085   CALL FDV$RETCX( TCA%, WKSP%, FRMNAM$, UARVAL$, CURPOS%, FLDTRM%, INSOVR%, HELPNUM% )
18100
18105   !+
18110   ! Break up the list into numbers.  Check each against the actual
18115   ! terminator. If terminator found in list, return success.
18120   !-
18122   UARVAL$ = UARVAL$ + '  '
18125   NONBLANK% = 1                   ! Beginning of string
18130   WHILE SEG$( UARVAL$, NONBLANK%, NONBLANK% ) <> ' '
18135       NEXTBLANK% = POS( UARVAL$, ' ', NONBLANK% )
18140       IF FLDTRM% = VAL( SEG$( UARVAL$, NONBLANK%, NEXTBLANK% - 1 ) ) THEN
                PASSKY = FDV$K_UKEY_TRM         !Pass Key to application
                FUNCTIONEXIT
18150       NONBLANK% = NEXTBLANK% + 1
18155   NEXT
18160   PASSKY = FDV$K_UKEY_ERR             Let FDV do the beeping
18165   FUNCTIONEND
```

```
19000   FUNCTION INTEGER CHKCHK
19010   !+*********************************************************************
19015   ! UAR for SAMP CHECK form.  Makes sure that the check amount is
19020   ! less than or equal to the current balance.  If not, complain and
19023   ! change video attributes on balance field so the potential bouncer
19024   ! can see what there is to work with.
19025   !-*********************************************************************
19027   DECLARE INTEGER CONSTANT                                              &
                FDV$K_UVAL_SUC= 1000,   !Field completion success             &
                FDV$K_UVAL_FAIL=1001    !Field completion failure
19029   !+
19030   ! Don't forget to pre-extend BASIC string variables before FMS calls.
19035   !-
19040   BALANCE$ = SPACE$( 6 )
19045   AMTPAY$ = SPACE$( 6 )
19050   CALL FDV$RET( BALANCE$, 'BALANCE' )
19055   CALL FDV$RET( AMTPAY$, 'AMTPAY' )
19060   IF VAL( BALANCE$ ) >= VAL( AMTPAY$ ) THEN
            CHKCHK = FDV$K_UVAL_SUC
            BLINKBOLD% = -1                         !Restore to original
            CALL FDV$AFVA( BLINKBOLD%, 'BALANCE' )
        ELSE
            CHKCHK = FDV$K_UVAL_FAIL
            BLINKBOLD% = 3                          !Make it very visible
            CALL FDV$AFVA( BLINKBOLD%, 'BALANCE' )
            CALL FDV$PUTL( "Your balance doesn't cover that much, reenter amount" )
19080   FUNCTIONEND
```

```
20000   FUNCTION INTEGER RANGE
20010   !+*********************************************************************
20015   ! General purpose UAR to check the range of any numeric item.  The
20020   ! associated UAR data must have one of the four forms:
20025   !       L,U:space:{message}
20030   !       -U:space:{message}
20035   !       L,:space:{message}
20040   !       ,:space:{message}
20045   ! where L is lower bound, U is upper bound, and {message} is an
20050   ! optional error message in case the field value is out of bounds.
20055   ! If one of the bounds isn't given, it isn't checked for.  If neither
20060   ! bound is given, nothing is checked, everything succeeds.  If the
20065   ! UAR value doesn't have a comma, a FDV$_UAR error message is returned
20070   ! to the calling program by the FDV so the form designer has to go
20075   ! back and do it right. If no {message} is given, a simple
20080   ! "out of range U:_" message is given to the hapless operator.
20085   !
20090   ! This UAR can work with any form and numeric field since it gets
20095   ! context itself.  Care must be taken with fields using field marker
20097   ! periods since those periods are not returned to the program.
20098   !-*********************************************************************
20099   DECLARE INTEGER CONSTANT                                              &
            FDV$K_UVAL_SUC= 1000,   !Field completion success                 &
            FDV$K_UVAL_FAIL=1001    !Field completion failure
20100   !+
20102   ! Pre-extend the strings into which FMS will return values.
20105   ! Get context which yields associated data value (ignore other stuff).
20110   ! Get current field name and index.
20115   ! Get field value.
20120   !-
20125   FRMNAM$ = SPACE$(31)
20130   UARVAL$ = SPACE$(80)
20132   NAME$ = SPACE$(31)
20133   NUMBER$ = SPACE$(132)
20135   CALL FDV$RETCX( TCA%, WKSP%, FRMNAM$, UARVAL$, CURPOS%, FLDTRM%, INSOVR%, HELPNUM% )
20140   CALL FDV$RETFN( NAME$, INDEX% )
20145   CALL FDV$RET( NUMBER$, NAME$, INDEX% )
20150   NUMBER = VAL( NUMBER$ )
20155   !+
20157   ! Find comma and blank delimiters.
20160   ! Check for lower bound.
20165   !-
```

```
20170   COMMA% = POS( UARVAL$, ',' , 1 )
20172   BLANK% = POS( UARVAL$, SPACE$(1), COMMA% + 1 )
20175   IF COMMA% = 0 THEN
            RANGE = 0    ! Illegal UARVAL string, FDV returns error
            FUNCTIONEXIT
20180   IF COMMA% <> 1 THEN
            IF NUMBER < VAL( SEG$( UARVAL$, 1, COMMA% - 1 ) ) THEN 20300
20190
20195   !+
20200   ! Check for upper bound
20205   !-
20215   IF BLANK% <> COMMA% + 1 THEN
            IF NUMBER > VAL( SEG$( UARVAL$, COMMA% + 1, BLANK% - 1 ) ) THEN 20300
20220   !+
20225   ! Passed both tests successfully, return success for UAR value
20230   !
20235   RANGE = FDV$K_UVAL_SUC
20240   FUNCTIONEXIT
20300   !+
20305   ! Error in one of the bounds.
20310   ! Give error message: either from the UARVAL or make one up.
20315   !-
20320   IF SEG$( UARVAL$, BLANK% + 1, BLANK% + 1 ) <> SPACE$(1) THEN
            CALL FDV$PUTL( SEG$( UARVAL$, BLANK% + 1, 80 ) )
        ELSE
            CALL FDV$PUTL( 'Field value out of bounds. Must be in range "' + &
                           SEG$( UARVAL$, 1, BLANK% - 1 ) + '".' )
20325   CALL FDV$SIGOP              !Beep, too.
20330   RANGE = FDV$K_UVAL_FAIL
20335   FUNCTIONEND
```

```
!*****************************************************************************
! FDVDEF.BAS  --  This is the include file for FMS applications in BASIC
!*****************************************************************************

!************************
! FMS terminator codes:
!************************
DECLARE INTEGER CONSTANT FDV$K_FT_NTR      =  0  !Enter (i.e. end GETs)
DECLARE INTEGER CONSTANT FDV$K_FT_NXT      =  1  !Next field
DECLARE INTEGER CONSTANT FDV$K_FT_PRV      =  2  !Previous field
DECLARE INTEGER CONSTANT FDV$K_FT_ATB      =  3  !Automatically move to next field
DECLARE INTEGER CONSTANT FDV$K_FT_XBK      =  4  !Exit scrolled area backward
DECLARE INTEGER CONSTANT FDV$K_FT_XFW      =  5  !Exit scrolled area forward
DECLARE INTEGER CONSTANT FDV$K_FT_SNX      =  6  !Scroll forward to next field
DECLARE INTEGER CONSTANT FDV$K_FT_SPR      =  7  !Scroll backward to previous field
DECLARE INTEGER CONSTANT FDV$K_FT_SFW      =  8  !Scroll forward
DECLARE INTEGER CONSTANT FDV$K_FT_SBK      =  9  !Scroll backward
DECLARE INTEGER CONSTANT FDV$K_FT_ILG_NXT  = 11  !Illegal context for next field
DECLARE INTEGER CONSTANT FDV$K_FT_ILG_PRV  = 12  !Illegal context for previous field
DECLARE INTEGER CONSTANT FDV$K_FT_ILG_ATB  = 13  !Illegal context for auto move to next field
DECLARE INTEGER CONSTANT FDV$K_FT_ILG_XBK  = 14  !Illegal context for exit scrolled area backward
DECLARE INTEGER CONSTANT FDV$K_FT_ILG_XFW  = 15  !Illegal context for exit scrolled area forward
DECLARE INTEGER CONSTANT FDV$K_FT_ILG_SFW  = 16  !Illegal context for scroll forward
DECLARE INTEGER CONSTANT FDV$K_FT_ILG_SBK  = 17  !Illegal context for scroll backward
!*******************************************************
! Function key terminators returned from GETs and WAIT
! Also used as FDV keycodes for use with DFKBD.
!*******************************************************
DECLARE INTEGER CONSTANT FDV$K_AR_UP       =  99
DECLARE INTEGER CONSTANT FDV$K_AR_DOWN     = 100
DECLARE INTEGER CONSTANT FDV$K_AR_LEFT     = 101
DECLARE INTEGER CONSTANT FDV$K_AR_RIGHT    = 102
DECLARE INTEGER CONSTANT FDV$K_PF_1        = 103
DECLARE INTEGER CONSTANT FDV$K_PF_2        = 104
DECLARE INTEGER CONSTANT FDV$K_PF_3        = 105
DECLARE INTEGER CONSTANT FDV$K_PF_4        = 106
DECLARE INTEGER CONSTANT FDV$K_KP_NTR      = 107
DECLARE INTEGER CONSTANT FDV$K_KP_COM      = 108
DECLARE INTEGER CONSTANT FDV$K_KP_HYP      = 109
DECLARE INTEGER CONSTANT FDV$K_KP_PER      = 110
DECLARE INTEGER CONSTANT FDV$K_KP_0        = 112
DECLARE INTEGER CONSTANT FDV$K_KP_1        = 113
```

```
DECLARE INTEGER CONSTANT FDV$K_KP_2       =  114
DECLARE INTEGER CONSTANT FDV$K_KP_3       =  115
DECLARE INTEGER CONSTANT FDV$K_KP_4       =  116
DECLARE INTEGER CONSTANT FDV$K_KP_5       =  117
DECLARE INTEGER CONSTANT FDV$K_KP_6       =  118
DECLARE INTEGER CONSTANT FDV$K_KP_7       =  119
DECLARE INTEGER CONSTANT FDV$K_KP_8       =  120
DECLARE INTEGER CONSTANT FDV$K_KP_9       =  121
DECLARE INTEGER CONSTANT FDV$K_GAR_UP     =  227
DECLARE INTEGER CONSTANT FDV$K_GAR_DOWN   =  228
DECLARE INTEGER CONSTANT FDV$K_GAR_RIGHT  =  229
DECLARE INTEGER CONSTANT FDV$K_GAR_LEFT   =  230
DECLARE INTEGER CONSTANT FDV$K_GPF_1      =  231
DECLARE INTEGER CONSTANT FDV$K_GPF_2      =  232
DECLARE INTEGER CONSTANT FDV$K_GPF_3      =  233
DECLARE INTEGER CONSTANT FDV$K_GPF_4      =  234
DECLARE INTEGER CONSTANT FDV$K_GKP_NTR    =  235
DECLARE INTEGER CONSTANT FDV$K_GKP_COM    =  236
DECLARE INTEGER CONSTANT FDV$K_GKP_HYP    =  237
DECLARE INTEGER CONSTANT FDV$K_GKP_PER    =  238
DECLARE INTEGER CONSTANT FDV$K_GKP_0      =  240
DECLARE INTEGER CONSTANT FDV$K_GKP_1      =  241
DECLARE INTEGER CONSTANT FDV$K_GKP_2      =  242
DECLARE INTEGER CONSTANT FDV$K_GKP_3      =  243
DECLARE INTEGER CONSTANT FDV$K_GKP_4      =  244
DECLARE INTEGER CONSTANT FDV$K_GKP_5      =  245
DECLARE INTEGER CONSTANT FDV$K_GKP_6      =  246
DECLARE INTEGER CONSTANT FDV$K_GKP_7      =  247
DECLARE INTEGER CONSTANT FDV$K_GKP_8      =  248
DECLARE INTEGER CONSTANT FDV$K_GKP_9      =  249
!*****************************************
! FDV KeyFunctions. For use in DFKBD call.
!*****************************************
DECLARE INTEGER CONSTANT FDV$K_KF_GOLD  =   1
DECLARE INTEGER CONSTANT FDV$K_KF_RESET =   2
DECLARE INTEGER CONSTANT FDV$K_KF_CRSLF =   3
DECLARE INTEGER CONSTANT FDV$K_KF_CRSRT =   4
DECLARE INTEGER CONSTANT FDV$K_KF_DLCHR =   5
DECLARE INTEGER CONSTANT FDV$K_KF_DLFLD =   6
DECLARE INTEGER CONSTANT FDV$K_KF_INS   =   7
DECLARE INTEGER CONSTANT FDV$K_KF_OVR   =   8
DECLARE INTEGER CONSTANT FDV$K_KF_RFRSH =   9
```

```
DECLARE INTEGER CONSTANT FDV$K_KF_HELP   =  10
DECLARE INTEGER CONSTANT FDV$K_KF_NXT    =  11
DECLARE INTEGER CONSTANT FDV$K_KF_PRV    =  12
DECLARE INTEGER CONSTANT FDV$K_KF_NTR    =  13
DECLARE INTEGER CONSTANT FDV$K_KF_SBK    =  14
DECLARE INTEGER CONSTANT FDV$K_KF_SFW    =  15
DECLARE INTEGER CONSTANT FDV$K_KF_XBK    =  16
DECLARE INTEGER CONSTANT FDV$K_KF_XFW    =  17
DECLARE INTEGER CONSTANT FDV$K_KF_NONE   =   0
DECLARE INTEGER CONSTANT FDV$K_KF_DFLT   =  -1
!****************************************************************
! UAR return codes. These codes are returned by UAR to FDV.
!****************************************************************
! Field completion return codes
!******************************
DECLARE INTEGER CONSTANT FDV$K_UVAL_SUC =  1000 !Field completion success
DECLARE INTEGER CONSTANT FDV$K_UVAL_FAIL=  1001 !Field completion failure
DECLARE INTEGER CONSTANT FDV$K_UVAL_END =  1002 !Field completion suc-stop UARs
!************************
! Help UAR return codes
!************************
DECLARE INTEGER CONSTANT FDV$K_UHELP_NO =  2000 !No help given, try next step
DECLARE INTEGER CONSTANT FDV$K_UHELPED  =  2001 !Help given, continue sequence
DECLARE INTEGER CONSTANT FDV$K_UHELP_ALL=  2002 !Help given, repeat UAR
!******************************
! Function Key UAR return codes
!******************************
DECLARE INTEGER CONSTANT FDV$K_UKEY_ERR =  3000 !Fn Key failure, FDV signals
DECLARE INTEGER CONSTANT FDV$K_UKEY_TRM =  3001 !Fn Key success, normal F.K.
DECLARE INTEGER CONSTANT FDV$K_UKEY_NXT =  3002 !Fn Key succ, treat as NEXT
DECLARE INTEGER CONSTANT FDV$K_UKEY_NTR =  3003 !Fn Key succ, treat as ENTER
DECLARE INTEGER CONSTANT FDV$K_UKEY_SUC =  3004 !Fn Key succ, ignore
!****************************************************************************
! FDV status codes returned when FDV$... routines are called as functions.
! These codes are VMS status codes and can be signalled.  They correspond
! one-to-one with the FMS status codes retrievable from FDV$STAT.
!****************************************************************************
DECLARE INTEGER CONSTANT FDV$_SUC = 2719889
DECLARE INTEGER CONSTANT FDV$_INC = 2719897
DECLARE INTEGER CONSTANT FDV$_MOD = 2719905
DECLARE INTEGER CONSTANT FDV$_IMP = 2719922
DECLARE INTEGER CONSTANT FDV$_FSP = 2719930
```

```
DECLARE INTEGER CONSTANT FDV$_IOL = 2719938
DECLARE INTEGER CONSTANT FDV$_FLB = 2719946
DECLARE INTEGER CONSTANT FDV$_ICH = 2719954
DECLARE INTEGER CONSTANT FDV$_FCH = 2719962
DECLARE INTEGER CONSTANT FDV$_FRM = 2719970
DECLARE INTEGER CONSTANT FDV$_FNM = 2719978
DECLARE INTEGER CONSTANT FDV$_LIN = 2719986
DECLARE INTEGER CONSTANT FDV$_FLD = 2719994
DECLARE INTEGER CONSTANT FDV$_NOF = 2720002
DECLARE INTEGER CONSTANT FDV$_DSP = 2720010
DECLARE INTEGER CONSTANT FDV$_NSC = 2720018
DECLARE INTEGER CONSTANT FDV$_DNM = 2720026
DECLARE INTEGER CONSTANT FDV$_DLN = 2720034
DECLARE INTEGER CONSTANT FDV$_UTR = 2720042
DECLARE INTEGER CONSTANT FDV$_IOR = 2720050
DECLARE INTEGER CONSTANT FDV$_IFN = 2720058
DECLARE INTEGER CONSTANT FDV$_ARG = 2720066
DECLARE INTEGER CONSTANT FDV$_INI = 2720074
DECLARE INTEGER CONSTANT FDV$_STR = 2720082
DECLARE INTEGER CONSTANT FDV$_IVM = 2720090
DECLARE INTEGER CONSTANT FDV$_FVM = 2720098
DECLARE INTEGER CONSTANT FDV$_ITT = 2720106
DECLARE INTEGER CONSTANT FDV$_TCA = 2720114
DECLARE INTEGER CONSTANT FDV$_STA = 2720122
DECLARE INTEGER CONSTANT FDV$_WID = 2720130
DECLARE INTEGER CONSTANT FDV$_NFL = 2720138
DECLARE INTEGER CONSTANT FDV$_IBF = 2720146
DECLARE INTEGER CONSTANT FDV$_NDS = 2720154
DECLARE INTEGER CONSTANT FDV$_UDP = 2720162
DECLARE INTEGER CONSTANT FDV$_UAR = 2720170
DECLARE INTEGER CONSTANT FDV$_UNF = 2720178
DECLARE INTEGER CONSTANT FDV$_CAN = 2720194
DECLARE INTEGER CONSTANT FDV$_KIF = 2720202
DECLARE INTEGER CONSTANT FDV$_KEX = 2720210
DECLARE INTEGER CONSTANT FDV$_KTW = 2720218
DECLARE INTEGER CONSTANT FDV$_KIL = 2720226
DECLARE INTEGER CONSTANT FDV$_TMO = 2720234
DECLARE INTEGER CONSTANT FDV$_LLI = 2720242
DECLARE INTEGER CONSTANT FDV$_VAL = 2720250
DECLARE INTEGER CONSTANT FDV$_IFU = 2720258
DECLARE INTEGER CONSTANT FDV$_SYS = 2720266
```

```
!*****************************************************************************
! FMS status codes returned when FDV$STAT routine is called.
!*****************************************************************************
! Success codes.

DECLARE INTEGER CONSTANT FDV$K_SUC =     1
DECLARE INTEGER CONSTANT FDV$K_INC =     2
DECLARE INTEGER CONSTANT FDV$K_MOD =     3

! Failure codes

DECLARE INTEGER CONSTANT FDV$K_IMP =     -2
DECLARE INTEGER CONSTANT FDV$K_FSP =     -3
DECLARE INTEGER CONSTANT FDV$K_IOL =     -4
DECLARE INTEGER CONSTANT FDV$K_FLB =     -5
DECLARE INTEGER CONSTANT FDV$K_ICH =     -6
DECLARE INTEGER CONSTANT FDV$K_FCH =     -7
DECLARE INTEGER CONSTANT FDV$K_FRM =     -8
DECLARE INTEGER CONSTANT FDV$K_FNM =     -9
DECLARE INTEGER CONSTANT FDV$K_LIN =     -10
DECLARE INTEGER CONSTANT FDV$K_FLD =     -11
DECLARE INTEGER CONSTANT FDV$K_NOF =     -12
DECLARE INTEGER CONSTANT FDV$K_DSP =     -13
DECLARE INTEGER CONSTANT FDV$K_NSC =     -14
DECLARE INTEGER CONSTANT FDV$K_DNM =     -15
DECLARE INTEGER CONSTANT FDV$K_DLN =     -16
DECLARE INTEGER CONSTANT FDV$K_UTR =     -17
DECLARE INTEGER CONSTANT FDV$K_IOR =     -18
DECLARE INTEGER CONSTANT FDV$K_IFN =     -19
DECLARE INTEGER CONSTANT FDV$K_ARG =     -20
DECLARE INTEGER CONSTANT FDV$K_INI =     -21
DECLARE INTEGER CONSTANT FDV$K_STR =     -22
DECLARE INTEGER CONSTANT FDV$K_FVM =     -23
DECLARE INTEGER CONSTANT FDV$K_IVM =     -24
DECLARE INTEGER CONSTANT FDV$K_ITT =     -25
DECLARE INTEGER CONSTANT FDV$K_TCA =     -26
DECLARE INTEGER CONSTANT FDV$K_STA =     -27
DECLARE INTEGER CONSTANT FDV$K_WID =     -28
DECLARE INTEGER CONSTANT FDV$K_NFL =     -29
DECLARE INTEGER CONSTANT FDV$K_IBF =     -30
DECLARE INTEGER CONSTANT FDV$K_NDS =     -31
DECLARE INTEGER CONSTANT FDV$K_UDP =     -33
```

```
DECLARE INTEGER CONSTANT FDV$K_UAR =     -34
DECLARE INTEGER CONSTANT FDV$K_UNF =     -35
DECLARE INTEGER CONSTANT FDV$K_CAN =     -39
DECLARE INTEGER CONSTANT FDV$K_KIF =     -40
DECLARE INTEGER CONSTANT FDV$K_KEX =     -41
DECLARE INTEGER CONSTANT FDV$K_KTW =     -42
DECLARE INTEGER CONSTANT FDV$K_KIL =     -43
DECLARE INTEGER CONSTANT FDV$K_TMO =     -44
DECLARE INTEGER CONSTANT FDV$K_LLI =     -45
DECLARE INTEGER CONSTANT FDV$K_VAL =     -47
DECLARE INTEGER CONSTANT FDV$K_IFU =     -48
DECLARE INTEGER CONSTANT FDV$K_SYS =     -49
!*********************************************************
! Declare the FDV routines
!*********************************************************
EXTERNAL LONG FUNCTION FDV$ADLVA
EXTERNAL LONG FUNCTION FDV$AFVA
EXTERNAL LONG FUNCTION FDV$ATERM
EXTERNAL LONG FUNCTION FDV$AWKSP
EXTERNAL LONG FUNCTION FDV$BELL
EXTERNAL LONG FUNCTION FDV$CANCL
EXTERNAL LONG FUNCTION FDV$CDISP
EXTERNAL LONG FUNCTION FDV$CLEAR
EXTERNAL LONG FUNCTION FDV$DEL
EXTERNAL LONG FUNCTION FDV$DFKBD
EXTERNAL LONG FUNCTION FDV$DISP
EXTERNAL LONG FUNCTION FDV$DPCOM
EXTERNAL LONG FUNCTION FDV$DTERM
EXTERNAL LONG FUNCTION FDV$DWKSP
EXTERNAL LONG FUNCTION FDV$GET
EXTERNAL LONG FUNCTION FDV$GETAF
EXTERNAL LONG FUNCTION FDV$GETAL
EXTERNAL LONG FUNCTION FDV$GETDL
EXTERNAL LONG FUNCTION FDV$GETSC
EXTERNAL LONG FUNCTION FDV$ILTRM
EXTERNAL LONG FUNCTION FDV$LCHAN
EXTERNAL LONG FUNCTION FDV$LCLOS
EXTERNAL LONG FUNCTION FDV$LEDOF
EXTERNAL LONG FUNCTION FDV$LEDON
EXTERNAL LONG FUNCTION FDV$LOAD
EXTERNAL LONG FUNCTION FDV$LOPEN
EXTERNAL LONG FUNCTION FDV$NDISP
```

```
EXTERNAL LONG FUNCTION FDV$PFT
EXTERNAL LONG FUNCTION FDV$PUT
EXTERNAL LONG FUNCTION FDV$PUTAL
EXTERNAL LONG FUNCTION FDV$PUTD
EXTERNAL LONG FUNCTION FDV$PUTDA
EXTERNAL LONG FUNCTION FDV$PUTL
EXTERNAL LONG FUNCTION FDV$PUTSC
EXTERNAL LONG FUNCTION FDV$READ
EXTERNAL LONG FUNCTION FDV$RET
EXTERNAL LONG FUNCTION FDV$RETAL
EXTERNAL LONG FUNCTION FDV$RETCX
EXTERNAL LONG FUNCTION FDV$RETDI
EXTERNAL LONG FUNCTION FDV$RETDN
EXTERNAL LONG FUNCTION FDV$RETFL
EXTERNAL LONG FUNCTION FDV$RETFN
EXTERNAL LONG FUNCTION FDV$RETFO
EXTERNAL LONG FUNCTION FDV$RETLE
EXTERNAL LONG FUNCTION FDV$RFRSH
EXTERNAL LONG FUNCTION FDV$SIGOP
EXTERNAL LONG FUNCTION FDV$SPADA
EXTERNAL LONG FUNCTION FDV$SPOFF
EXTERNAL LONG FUNCTION FDV$SPON
EXTERNAL LONG FUNCTION FDV$SSIGQ
EXTERNAL LONG FUNCTION FDV$SSRV
EXTERNAL LONG FUNCTION FDV$STAT
EXTERNAL LONG FUNCTION FDV$SWKSP
EXTERNAL LONG FUNCTION FDV$WAIT
```

Chapter 3
Programming FMS Applications in VAX–11 BLISS–32

The FMS Form Driver processes all Form Driver calls according to VAX–11 standards and language-specific rules. These rules define how parameters are passed to the Form Driver and how values are returned to your program. Language-specific information is briefly presented in this manual. For more detail, refer to the VAX–11 BLISS document set.

Your VAX–11 BLISS application program must comply with the requirements of the VAX–11 BLISS FMS interface. Topics discussed in this chapter include:

- Form Driver Routines
  - Invoking Form Driver Routines as Procedures
  - Accessing Form Driver Status Codes as Functions
- Parameter Passing in FMS
- Null Arguments
- FMS Data Types
  - Character Strings
  - Longword Binary Integers
  - Word Binary Integers
- Non-FMS Data Types
- One-Dimensional Arrays (Vectors)
- Allocation: Workspace, Terminal Control Area, and Run-Time Memory-Resident Form Area
- Precautions for Using FMS
- Data Conversion
- Sample Application Program in VAX–11 BLISS–32

A sample program written in BLISS (SAMPBLI.BLI) appears at the end of this chapter. Following the code for SAMPBLI.BLI are Form Driver REQUIRE files created for the Sample Application. Command file information needed to build the Sample Application program is in Section 3.10.2.

Examples from the Sample Application are used throughout the text to illustrate language issues. Where appropriate examples from SAMPBLI.BLI do not exist, other examples are provided.

3.1 Form Driver Routines

You can call any FMS routine as a procedure or as a function. Syntax follows standard VAX–11 BLISS–32 requirements.

3.1.1 Invoking Form Driver Routines as Procedures

You use the procedure call statement to invoke an FMS Form Driver routine. For example:

```
FDV$WAIT  ();
```

Calls the Form Driver routine FDV$WAIT and passes no parameters.

```
FDV$GET  (OPTION_DSC, TERMINATOR, %ASCID'OPTION');
```

Calls the Form Driver routine FDV$GET and passes three parameters.

See Appendix A for a complete list of Form Driver calls. The calling sequence for each Form Driver procedure, data access codes, data types, and passing mechanisms are presented in language-independent notation as specified by the VAX–11 Procedure Calling and Condition Handling Standard. For further detail about the VAX–11 Procedure Calling and Condition Handling Standard, refer to the *VAX–11 Run-Time Library Reference Manual.*

3.1.2 Accessing Form Driver Status Codes as Functions

An FMS status code is returned to the calling program at the completion of all Form Driver calls. To receive the returned status code from a Form Driver routine, you activate the routine with a function reference. Note that this returns a standard VMS status code. For portability, other status mechanisms can also be used. (For more information, see the *VAX–11 FMS Form Driver Reference Manual*, Chapter 2.)

The following statement calls FDV$GET as an FMS function:

```
STATUS_RETURN=FDV$GET (OPTION_DSC, TERMINATOR, %ASCCID'OPTION');
```

3.2 Parameter Passing in FMS

The parameter passing mechanism refers to the way in which data is passed to a called routine. The VAX–11 Procedure Calling Standard has three methods for passing parameters:

- By reference
- By descriptor
- By value

FMS routines, however, expect parameters to be passed only by reference and by descriptor.

By reference specifies that the storage location of the parameter is passed to the routine. FMS expects integers to be passed by reference.

By descriptor specifies that the address of a descriptor data structure is passed to the called routine. FMS expects character strings and arrays to be passed by descriptor.

3.3 Null Arguments

When the call syntax includes optional parameters and you do not wish to specify all of the information, you can use null arguments. Any optional parameter can be omitted to simplify your program. An address of 0 is assigned to each null argument. Optional parameters to the right of the last required parameter can simply be omitted from the call. In the following example, the FDV$GETAL call passes only the field terminator value:

```
FDV$GETAL  (0, TERMINATOR);
```

3.4 FMS Data Types

3.4.1 Character Strings

The character string is one of the general data types used by FMS. For example, the FDV$GET call passes the character strings for field value (OPTION_DSC) and field name ('OPTION'):

```
FDV$GET  (OPTION_DSC, TERMINATOR, %ASCID'OPTION');
```

BLISS does not support character strings explicitly with the exception of %ASCID literals. BLISS does, however, have data structures that you can use as character strings. Because BLISS does not support string descriptors implicitly, you must create your own. String descriptor data structures can be declared using macros defined in the system library.

When you use descriptors, be certain that your strings are initially declared to be long enough to accommodate your FMS data. Although FMS accepts both dynamic and fixed-length strings as parameters, it treats all strings as type FIXED. In other words, FMS does not alter the length of a dynamic string descriptor when the Form Driver returns values to the output arguments.

Two approaches are available for satisfying the fixed-length string constraints of FMS. One option is to declare your fixed-length strings to be the exact length of the FMS data to be returned. You can use the FMS/DESCRIPTION/BRIEF command to determine the length of the strings.

Alternatively, a single string variable can be used in different FMS calls to transfer data to or from several forms and fields. You must declare the string variable to be at least as large as the longest field value string that will be returned to your program. You can also use the FMS/DESCRIPTION/BRIEF command to access this information. Use the FDV$RETLE call to return the length of the valid portion of the field value in the string variable. This length can then be used when referencing the data that has been entered in the field.

```
FDV$GET (ACCOUNT, TERMINATOR, %ASCID'FIELD');
FDV$RETLE (LENGTHFIELD, %ASCID'FIELD');
```

After the execution of the FDV$RETLE call, LENGTHFIELD is equal to the length of the field named 'FIELD'. It is also equal to the valid portion of the string that is defined by the string descriptor ACCOUNT. LENGTHFIELD is now used to reference the data that was entered in the field named 'FIELD'.

A useful application of the FDV$RETLE call is in general purpose user action routines.

3.4.2 Longword Binary Integers

The longword binary integer is another general data type used by FMS. For example, the FDV$ATERM call passes the longword value for terminal control area size (12) and logical I/O channel number (2):

```
FDV$ATERM  (TCA_DSC, %REF(12), %REF(2));
```

Numeric parameters must be longword signed binary integers. If you try to pass other numeric types to the Form Driver, the calls do not work properly. An exception is the FDV$DFKBD call (see the following section).

3.4.3 Word Binary Integers

The defkbd parameter is a word integer array passed when the FDV$DFKBD routine is called. FMS expects a word integer array to be passed by descriptor.

3.5 Non-FMS Data Types

BLISS data types that are not recognized by FMS can be used in your BLISS application program provided they are not passed to the Form Driver.

3.6 One-Dimensional Arrays (Vectors)

One-dimensional arrays (vectors) are structures that can be used in FMS for the following parameters:

- tca (terminal control area)
- wksp (workspace)
- mloc (memory location)
- defkbd (define keyboard)

You must provide FMS with storage space for these parameters. You can do that by defining them to be:

- longword integer arrays or character strings for tca, wksp, and mloc
- word integer arrays for defkbd

In the Sample Application program, the tca, wksp, and mloc parameters are passed to several Form Driver routines. These parameters are defined as integer array descriptors. In the program a distinct macro is constructed for declaring longword integer arrays and their descriptors that are passed by the Form Driver.

The following declarations establish names and storage for the integer array variables WORKSPACE, CHECKWKSP, TCA, and MENU_FORM. L_ARRAY is a macro in the SAMPBLI.BLI program.

```
L_ARRAY(
    WORKSPACE,          3,      !General workspace
    CHECKWKSP,          3,      !Check workspace
    TCA,                3,      !Terminal Control Area
    MENU_FORM,        500,      !Storage for memory-resident forms
    );
```

You can alternatively define these variables to be character strings. (The strings can be static or dynamic but must be extended to the proper length). Use of character strings rather than longword integer arrays avoids the need to construct a distinct macro for the arrays. Thus, as in the example below, you may wish to declare tca, wksp, and mloc as fixed-length character strings. FIXSTR is a macro in the SAMPBLI.BLI program.

```
FIXSTR(
    WORKSPACE,     12,
    CHECKWKSP,     12,
    TCA,           12,
    MENU_FORM,   2000
    );
```

3.7 Allocation: Workspace, Terminal Control Area, and Run-Time Memory-Resident Form Area

The FMS workspace, terminal control area, and run-time memory-resident form area variables are character strings or longword integer arrays. These variables should be placed in OWN or GLOBAL. Note that this is not done in the Sample Application program. The sample program's structure pro-

tects the workspaces, terminal control areas, and run-time memory-resident form areas implicitly.

The Form Driver needs 12 bytes to associate user information about each workspace and terminal control area. The allocation for a run-time memory-resident form area must be the size of the form. The space for all of these variables is allocated by your application program. Note that FMS uses *only* 12 bytes of space for the workspace and terminal control area allocation. Any allocation greater than 12 bytes is wasted. You declare your workspace, terminal control area, and run-time memory-resident form area only once. FMS remembers their addresses after the addresses have been initially passed to the Form Driver.

For each workspace, the Form Driver also allocates an additional amount of storage space based on your estimate of the amount of memory needed to store your largest form. If your estimate is too small, the Form Driver allocates more space automatically, but performance may be affected. An adequate estimate results in more efficient operation of the Form Driver. You can use the FMS/DIRECTORY/FULL command to find out how much space to allocate.

In the following example from the Sample Application program, workspace is allocated and the FDV$AWKSP routine is called. In the program's declaration section, 12 bytes (3 longwords) are allocated. When FDV$AWKSP is called, the first parameter (WORKSPACE_DSC) specifies the area of memory to be used for your workspace. The second parameter specifies an estimate of the workspace size (2000 bytes) that you will need to display the largest form in your application.

```
MACRO
    L_ARRAY[NAME, LENGTH] =
     OWN
          %NAME(NAME, '_PTR') : VECTOR[LENGTH],
          %NAME(NAME, '_DSC') : L_ARRAYDSC(LENGTH, %NAME(NAME, '_PTR'));
    LITERAL
          %NAME(NAME, '_LEN') = LENGTH %;
MACRO
    L_ARRAYDSC(LEN, PTR) =
          BLOCK[16, BYTE]
             PRESET(
             [DSC$W_LENGTH] = 4,                    ! Length in bytes of an element
                   [DSC$B_DTYPE] = DSC$K_DTYPE_L,   ! Longword integer
                   [DSC$B_CLASS] = DSC$K_CLASS_A,   ! Array
                   [DSC$B_DIMCT] = 1                ! Number of dimensions
%IF NOT %NULL(PTR) %THEN
          ,[DSC$A_POINTER] = PTR                    ! Address of first byte
%FI
%IF NOT %NULL(LEN) %THEN
          ,[DSC$L_ARSIZE] = LEN * 4                 ! Total size of array in bytes
%FI
          ) %;
OWN
L_ARRAY(
    WORKSPACE,      3,          !General workspace
    CHECKWKSP,      3,          !Check workspace
    );
FDV$AWKSP (WORKSPACE_DSC, %REF(2000));
FDV$AWKSP (CHECKWKSP_DSC, %REF(2000));
```

3.8 Precautions for Using FMS

3.8.1 Memory Areas Used Exclusively by FMS

The locations for terminal control area, workspace, and run-time memory-resident form area are used exclusively by FMS. The terminal control area and workspace are attached with the FDV$ATERM and FDV$AWKSP calls and remain allocated until the FDV$DTERM and FDV$DWKSP calls are issued or until the program ends. The run-time memory-resident form area, used in the FDV$READ call, remains allocated until the FDV$DEL call is issued or until the program ends. You never touch the terminal control areas, workspaces, or run-time memory-resident form areas in your program except to pass their addresses to the Form Driver.

3.8.2 Why You Should Use the OWN or GLOBAL Attribute

Parameters to the following Form Driver routines should be used with caution:

FDV$ATERM	Attach terminal
FDV$AWKSP	Attach form workspace
FDV$READ	Read form into memory
FDV$SSRV	Specify status reporting variables

For example, once an FDV$SSRV call is issued, the program variables that contain the FMS status and RMS status become volatile and can change at any call point. As a general rule, you should place the status variables in OWN or GLOBAL storage.

In cases where you need both the FMS and RMS statuses, the FDV$STAT routine can be used. Note that only the FDV$STAT and FDV$SSRV calls provide RMS status. With the FDV$STAT routine, you do not have to worry about volatility.

The locations for terminal control area, workspace, run-time memory-resident form area, and status reporting variables must all continue to exist while the Form Driver is using them. They must remain allocated until the terminal control area and workspace are detached, until forms in memory location are deleted and until the status reporting variables are no longer used. LOCAL is only allocated while the current routine is active.

3.8.3 Using the Form Driver as a Shareable Image

To use the Form Driver as a shareable image, you must set the addressing mode to EXTERNAL = GENERAL. This is necessary because the default does not generate position-independent references that are required to link with the Form Driver as a shareable image. You can include the following in your program module header:

```
%BLISS 32 (,ADDRESSING_MODE (EXTERNAL = GENERAL))
```

3.9 Data Conversion

FMS uses only ASCII character strings to display data. All information displayed on the terminal screen, and all information received from the terminal operator, is represented as ASCII field values. Any manipulation of numeric data requires conversion of ASCII character strings to numeric data, and conversion of numeric data back to ASCII character strings. In the following discussion of conversion routines, you should assume that the receiving data type can support the largest number that is likely to be generated.

You can create your own data conversion functions to help satisfy FMS requirements. The Sample Application has the following data conversion macros:

```
KEYWORDMACRO
    VAL(LEN, PTR, DSC) =
     !+
     ! Given a string containing ASCII digits as a length and
     ! pointer or as a descriptor, return the numeric value as
     ! a longword.
     !-
     %IF %NULL(DSC)
     %THEN
         BEGIN
         EXTERNAL ROUTINE BAS$VAL_L : ADDRESSING_MODE(GENERAL);
         OWN STR_ : FIXDSC(%IF %CTCE(LEN) %THEN LEN %FI);
         %IF NOT %CTCE(LEN) %THEN STR_[DSC$W_LENGTH] = LEN; %FI
         STR_[DSC$A_POINTER] = PTR;
         BAS$VAL_L(STR_)
         END
     %ELSE
         BEGIN
         EXTERNAL ROUTINE BAS$VAL_L : ADDRESSING_MODE(GENERAL);
         BAS$VAL_L(DSC)
         END
     %FI %;

MACRO
    STR(INPUT_VAL) =
     !+
     ! Given a longword value return the corresponding
     ! ASCII decimal string as a descriptor.
     !-
     BEGIN
     EXTERNAL ROUTINE BAS$STR_L : ADDRESSING_MODE(GENERAL);
     OWN STR_ : DYNDSC;
     BAS$STR_L(STR_, INPUT_VAL);
     STR_
     END %,
```

In the Sample Application, the following steps are taken to get a new account balance after writing a check:

```
FDV$RET (RI_AMTPAY_DSC, %ASCID'AMTPAY');
AMTPAY = VAL(DSC = RI_AMTPAY_DSC);
BALANCE = .BALANCE - .AMTPAY;
TOTPAY = .TOTPAY + .AMTPAY;

FDV$PUT (STR$(.BALANCE), %ASCID'BALANCE');
```

In this example, the keyword macro VAL reads a string containing ASCII digits as a descriptor and returns the numeric value as a longword integer. The integer value of the variable AMTPAY can now be subtracted from the integer value of the variable BALANCE to produce a new value for BALANCE. AMTPAY can also be added to the integer value of the variable TOTPAY to produce a new value for TOTPAY.

After the data operations have been completed, the macro STR converts the longword integer value of the variable BALANCE to the corresponding ASCII decimal string descriptor. The value for the balance is displayed in the right-justified field 'BALANCE'. The rightmost digit from the program is displayed in the field's rightmost character position. The remaining digits of the ASCII decimal string are placed to the left of the rightmost digit. If output is longer than the field, FMS truncates on the left. (The Form Driver displays a data length error message (FDV$_DLN) only if you have set FMS Debug mode.)

For other conversion options, see the general conversion routines in the *VAX–11 Run–Time Library Reference Manual.*

3.10 Sample Application Program in VAX–11 BLISS–32

The FMS Sample Application program (SAMPBLI.BLI) is part of the FMS distribution kit. When FMS is installed, SAMPBLI.BLI is placed in the directory FMS$EXAMPLES. Designed to be a demonstration program and learning tool, SAMPBLI.BLI shows most of the features provided by FMS. The entire Sample Application program appears at the end of this chapter.

3.10.1 Form Driver Definition Files

The file FDVDEF.REQ is part of the Sample Application program package. When FMS is installed, FDVDEF.REQ is placed in the directory FMS$EXAMPLES. The FDVDEF.REQ file appears after the Sample Application source code.

FDVDEF.REQ contains a variety of codes for the Form Driver routines used in the Sample Application program. Although these codes have been created for use in SAMPBLI.BLI, they can provide you with a helpful starting point as you create definitions for your own application program. The file FDVDEF.REQ includes:

- FMS terminator codes
- Function key terminators returned from the FDV$GET and FDV$WAIT calls
- Form Driver key functions for use with the FDV$DFKBD call
- User action routine (UAR) return codes, which are returned by the UARs to the Form Driver:

  Field completion UAR return codes

  Help UAR return codes

  Function key UAR return codes

- VMS status codes returned when Form Driver routines are called as functions. These codes can be signaled.
- FMS status codes returned when the FDV$STAT routine is called as a function
- Declarations of Form Driver routines

3.10.2 Command File for Building the Sample Application Program

The command file for building the Sample Application program includes all the information that you need to compile and link SAMPBLI.BLI. When FMS is installed, the command file is placed in the directory FMS$EXAMPLES.

```
$!     S A M P B L I . C O M
$!
$!     Compile and link the BLISS version of the FMS V2 Sample Application
$!
$!     The BLISS source files are:      SAMPBLI.BLI
$!                                      FDVDEF.BLI
$!
$!     SMPVECTOR.OBJ and SMPMEMRES.OBJ were produced by the FMS commands:
$!
$!       $ FMS/VECTOR/OUTPUT=SMPVECTOR SAMP.FLB
$!       $ FMS/MEMORY/OUTPUT=SMPMEMRES SAMP.FLB/FORM=(HELP_KEYS,HELP_MENU)
$!
$ BLISS SAMPBLI
$ LINK  SAMPBLI, FMS$EXAMPLES:SMPVECTOR, FMS$EXAMPLES:SMPMEMRES
```

```
MODULE SAMP     (MAIN = SAMP %TITLE 'The FMS Sample Application Program'
                %BLISS32 (, ADDRESSING_MODE ( EXTERNAL = GENERAL ) )
                ) =
BEGIN
!+
!
! SAMP -- The FMS Sample Application Program
!
!-
LIBRARY 'SYS$LIBRARY:STARLET';

!+
! Table of Contents
!-
FORWARD ROUTINE
    INACCT      : NOVALUE,
    FMTCHK      : NOVALUE,
    MENU        : NOVALUE,
    MRITCH      : NOVALUE,
    ONECHK      : NOVALUE,
    ENDCHK      : NOVALUE,
    PRICHK      : NOVALUE,
    MAKDEP      : NOVALUE,
    VUEREG      : NOVALUE,
    SCRFWD      : NOVALUE,
    SCRBAK      : NOVALUE,
    VUEACT      : NOVALUE,
    GETAL       : NOVALUE,
    GETSTA      : NOVALUE,
    SRVCHK      : NOVALUE,
    ABORT       : NOVALUE,
    VALID1,
    TAKE15,
    PASSKY,
    CHKCHK,
    RANGE;

EXTERNAL ROUTINE
    LIB$SYS_FAO :       ADDRESSING_MODE(GENERAL),
    LIB$PUT_OUTPUT :    ADDRESSING_MODE(GENERAL),
    STR$TRIM :          ADDRESSING_MODE(GENERAL);
```

```
!+
! Macros for declaring descriptors for fixed and dynamic strings and for
! declaring a fixed descriptor and storage.
!
! Examples:
!
! OWN
!     X : DYNDSC,          ! Declares an uninitialized dynamic descriptor.
!     Y : FIXDSC(8,BF);    ! Declares a fixed descriptor (Length=8,Pointer=BF).
! FIXSTR(
!     Z, 20);              ! Allocates a fixed string of length 20 in OWN storage.
!                          ! It declares Z_LEN (length), Z_PTR (pointer), and
!                          ! Z_DSC (fixed descriptor).
!-
MACRO
    DYNDSC =
        BLOCK[DSC$K_S_BLN, BYTE]
                PRESET(
                [DSC$W_LENGTH] = 0,              ! Length in bytes
                [DSC$B_DTYPE] = DSC$K_DTYPE_T,   ! ASCII Text String
                [DSC$B_CLASS] = DSC$K_CLASS_D,   ! Dynamically Allocated String
                [DSC$A_POINTER] = 0              ! Address of first byte
                ) %,

    FIXDSC( LEN, PTR ) =
        BLOCK[DSC$K_S_BLN, BYTE]
                PRESET(
%IF NOT %NULL(LEN) %THEN
                [DSC$W_LENGTH] = LEN,            ! Length in bytes
%FI
                [DSC$B_DTYPE] = DSC$K_DTYPE_T,   ! ASCII Text String
                [DSC$B_CLASS] = DSC$K_CLASS_S    ! Scalar or Fixed Length String
%IF NOT %NULL(PTR) %THEN
                ,[DSC$A_POINTER] = PTR           ! Address of first byte
%FI
                ) %,
```

```
    FIXSTR[ NAME, LENGTH ] =
        OWN
            %NAME( NAME, '_PTR' ) : VECTOR[ LENGTH, BYTE],
            %NAME( NAME, '_DSC' ) : FIXDSC( LENGTH, %NAME( NAME, '_PTR') );
        LITERAL
            %NAME( NAME, '_LEN' ) = LENGTH %;

!+
! Pointers to a few strings.
!-
BIND
    BLANK_PTR = CH$PTR(UPLIT(' ')),
    COMMABLANK_PTR =    CH$PTR(UPLIT(', ')),
    PERIODBLANK_PTR =   CH$PTR(UPLIT('. '));
!+
! Macros for declaring longword integer arrays.
!
! Note that these are similar to the macros for fixed strings,
! but that the lengths specified are in terms of longwords instead of bytes.
!-
MACRO
    L_ARRAYDSC( LEN, PTR ) =
        BLOCK[16, BYTE]
                PRESET(
                [DSC$W_LENGTH] = 4,             ! Length in bytes of an element
                [DSC$B_DTYPE] = DSC$K_DTYPE_L,  ! Longword integer
                [DSC$B_CLASS] = DSC$K_CLASS_A,  ! Array
                [DSC$B_DIMCT] = 1               ! Number of dimensions
%IF NOT %NULL(PTR) %THEN
                ,[DSC$A_POINTER] = PTR          ! Address of first byte
%FI
%IF NOT %NULL(LEN) %THEN
                ,[DSC$L_ARSIZE] = LEN * 4       ! Total size of array in bytes
%FI
                ) %,

    L_ARRAY[ NAME, LENGTH ] =
        OWN
            %NAME( NAME, '_PTR' ) : VECTOR[ LENGTH ],
            %NAME( NAME, '_DSC' ) : L_ARRAYDSC( LENGTH, %NAME( NAME, '_PTR') );
        LITERAL
            %NAME( NAME, '_LEN' ) = LENGTH %;
```

```
!+
! Macros for declaring records.
!
! Example:
!
! DECL_REC(
!       A,
!           X,  5,
!           Y,  7 );
!
! Allocates a BLOCK structure A of length 12 with fields X_FLD and Y_FLD in
! fieldset A_FLDS.
!
! It also declares the following:
!       Literals:       A_LEN = 12, X_LEN = 5, Y_LEN = 7
!       Pointers:       A_PTR, X_PTR, Y_PTR
!       Descriptors:    A_DSC, X_DSC, Y_DSC
!-
MACRO
    DECL_REC( REC ) =
        %ASSIGN( DECL_REC_LEN, 0 )

        FIELD %NAME(REC,'_FLDS') =
            SET
            DECL_REC_FLDS( %REMAINING )
            TES;

        LITERAL
            DECL_REC_LENS( %REMAINING ),
            %NAME(REC,'_LEN') = DECL_REC_LEN;

        OWN
            REC : BLOCK[DECL_REC_LEN,BYTE] FIELD( %NAME(REC,'_FLDS') );

        BIND
            DECL_REC_PTRS( REC, %REMAINING ),
            %NAME(REC,'_PTR') = REC;

        OWN
            DECL_REC_DSCS( %REMAINING ),
            %NAME(REC,'_DSC') :
                FIXDSC( %NAME(REC,'_LEN') , %NAME(REC,'_PTR') ) %,
```

```
    DECL_REC_LENS[ NAME, LEN ] =
        %NAME(NAME,'_LEN') = LEN %,

    DECL_REC_FLDS[ NAME, LEN ] =
        %NAME(NAME,'_FLD') = [DECL_REC_LEN, 0, 0, 0]
        %ASSIGN( DECL_REC_LEN, DECL_REC_LEN + LEN ) %,

    DECL_REC_PTRS(REC)[ NAME, LEN ] =
        %NAME(NAME,'_PTR') = REC[%NAME(NAME,'_FLD')] %,

    DECL_REC_DSCS[ NAME, LEN ] =
        %NAME(NAME,'_DSC') : FIXDSC( %NAME(NAME,'_LEN'), %NAME(NAME,'_PTR') ) %;

COMPILETIME DECL_REC_LEN = 0;
KEYWORDMACRO
    VAL( LEN, PTR, DSC ) =
        !+
        ! Given a string containing ASCII digits as a length and pointer
        ! or as a descriptor, return the numeric value as a longword.
        !-
        %IF %NULL(DSC)
        %THEN
            BEGIN
            EXTERNAL ROUTINE BAS$VAL_L : ADDRESSING_MODE(GENERAL);
            OWN STR_ : FIXDSC(%IF %CTCE(LEN) %THEN LEN %FI);
            %IF NOT %CTCE(LEN) %THEN STR_[DSC$W_LENGTH] = LEN; %FI
            STR_[DSC$A_POINTER] = PTR;
            BAS$VAL_L( STR_ )
            END
        %ELSE
            BEGIN
            EXTERNAL ROUTINE BAS$VAL_L : ADDRESSING_MODE(GENERAL);
            BAS$VAL_L( DSC )
            END
        %FI %;
```

```
MACRO
    STR$( INPUT_VAL ) =
        !+
          Given a longword value return the corresponsing ASCII decimal string
        ! as a descriptor.
        !-
        BEGIN
        EXTERNAL ROUTINE BAS$STR_L : ADDRESSING_MODE(GENERAL);
        OWN STR_ : DYNDSC;
        BAS$STR_L( STR_, INPUT_VAL );
        STR_
        END %,

    TRM$( INPUT_STR ) =
        BEGIN
        EXTERNAL ROUTINE STR$TRIM : ADDRESSING_MODE(GENERAL);
        OWN STR_ : DYNDSC;
        STR$TRIM( STR_, INPUT_STR );
        STR_
        END %,

    SEG$( START, FINISH ) =
        !+
        ! Given two character pointers, return a descriptor for the character
        ! sequence that starts at START and goes up to but does not include
        ! the character at FINISH.
        !-
        BEGIN
        OWN STR_ : FIXDSC();
        STR_[DSC$W_LENGTH] = CH$DIFF((FINISH), (START));
        STR_[DSC$A_POINTER] = (START);
        STR_
        END %;
```

```
!+
! Account (Read in from File)
!-
DECL_REC(
    ACCOUNT,
        ACCTNO,         5,
        ACCDATE,        7,
        LAST,           20,
        FIRST,          15,
        MIDDLE,         15,
        STREET,         30,
        CITY,           20,
        STATE,          2,
        ZIP,            5,
        HOMEPH,         10,
        WORKPH,         10,
        OPW,            12 );

!+
! Deposit data (Read via FDV$GETAL)
!-
DECL_REC(
    DEPOSIT,
        DEP_DATE,       7,
        DEP_CURBAL,     6,
        DEP_AMT,        6,
        DEP_NEWBAL,     6,
        DEP_MEMO,       35 );
```

```
!+
! Money.
! Note that all money is kept internally as integers (in cents).
! It is only when the quantities are output that they look like
! dollars, since all the money fields have periods as field
! markers in the right places and they are right justified or
! fixed decimal.
!
! Register data.
! Define a single structure into which to put data via
! descriptor names and a vector of structures.  After data has been
! be put into the structure, it is copied to the array for convenience
! in scrolling.
!-
DECL_REC(
    REGITEM,
        RI_NUM,         4,
        RI_DATE,        7,
        RI_MEMPAYTO,    35,
        RI_AMTDEP,      6,
        RI_AMTPAY,      6,
        RI_BALANCE,     6 );

LITERAL
    REGSIZE = 30;

OWN
    REGARRAY:
        BLOCKVECTOR[ REGSIZE, REGITEM_LEN, BYTE ] FIELD (REGITEM_FLDS);

MACRO REGARRAY_DSC( I ) =
    BEGIN
    OWN
        STR_ : FIXDSC( REGITEM_LEN, 0 );
    STR_[DSC$A_POINTER] = REGARRAY[ I, 0,0,0,0];
    STR_
    END %;
```

```
!+
! Other variables
!-
OWN
    TERMINATOR,             ! Terminator returned by FDV
    BALANCE,                ! Balance in account, numeric
    SBALANCE,               ! Starting balance
    AMTPAY,                 ! Check payment amount
    TOTDEP,                 ! Total deposits made in this session
    TOTPAY,                 ! Total checks payed in this session
    FMSSTATUS :             ! Status for last FDV call
        INITIAL(1),
    RMSSTATUS :             ! RMS Status for last FDV call
        INITIAL(1),
    LASTREGNUM,             ! Last number used in the register (1...REGSIZE)
    LASTCHNUM,              ! Last check number used
    I,                      ! Index into lines of check
    NSCROL,                 ! "
    CURLINE,                ! Line of check register that cursor is now on
    MINWINDOW,              ! Smallest line of register being displayed
                            !       on the scrolled area
    MAXWINDOW,              ! Largest line of register being displayed
                            !       on the scrolled area
    FAKE_DSC : DYNDSC,      ! Value returned from fake field in scrolled area
    JUNK_DSC : DYNDSC,      ! Temporary storage for return from GETAL
    INSOVR,                 ! Insert/overstrike mode
    HELPNUM,                ! Number of help invocation
    FIELDNAME_DSC :         ! Name of field from FDV$GETAF
        DYNDSC;
FIXSTR(
    OPTION, 1,              ! Choice returned from menu
    FIRSTL, 2,              ! First line on the form of the check image
                            !       (from named data)
    LASTL, 2,               ! Last line on the form of the check image
                            !       (from named data)
    LINE, 80,               ! Line return as image of form for check print
    NSCROL, 2,              ! Number lines in scrolled area (from named data)
    PASSWORD, 12,           ! Password from account
    DONE, 80                ! Form done message for Deposit
    );
```

```
!+
! Data definitions
!
! FMS related
!-
LOCAL ARRAY(
    WORKSPACE,  3,      !General workspace
    CHECKWKSP,  3,      !Check workspace
    TCA,        5,      !Terminal Control Area
    MENU_FORM,  500,     Storage for memory resident forms
    CHECK_FORM, 750,
    DPOSIT_FORM, 500
    );

OWN
    SIZE_MENU,
    SIZE_CHECK,
    SIZE_DPOSIT;

!+
! FMS declarations.
!       Routines, codes, etc.
!-
REQUIRE 'FDVDEF.REQ' ;
```

```
ROUTINE SAMP =
    BEGIN
    !+
    ! Initialize FMS
    !   Attach default terminal
    !   Attach normal and check workspaces (order important for help
    !       and refresh during CHECK/CHECKDONE time--try switching and see).
    !   Open form library, attach to channel 1
    !   Set Keypad mode to application
    !   Set signal mode to bell (default, but it's fun to do)
    !-
    FDV$ATERM( TCA_DSC, %REF(12), %REF(2) );
    GETSTA();
    FDV$AWKSP( WORKSPACE_DSC, %REF( 2000 ) );
    GETSTA();
    FDV$AWKSP( CHECKWKSP_DSC, %REF( 2000 ) );
    GETSTA();
    FDV$LOPEN( %ASCID 'FMS$EXAMPLES:SAMP', %REF( 1 ) );
    GETSTA();
    FDV$SPADA( %REF( 1 ) );
    FDV$SSIGQ( %REF( 0 ) );

    !+
    ! Set all future calls to return status to the two status recording
    ! variables FMSSTATUS and RMSSTATUS without having to call the
    ! the FDV$STAT routine.
    !-
    FDV$SSRV( FMSSTATUS, RMSSTATUS );

    !+
    ! Read in a few forms from the form library onto the dynamic
    ! resident form list.  You may be able to detect the difference
    ! in the form to form access times for those forms which have to be
    ! accessed from the form library on disk and those forms which are
    ! on the dynamic or static memory resident form list.  See the
    ! installation notes for this program (the LINK command) to see
    ! which forms are on the static memory resident form list.
    !-
    FDV$READ( %ASCID 'MENU', MENU_FORM_DSC, %REF(2000), SIZE_MENU );
    FDV$READ( %ASCID 'CHECK', CHECK_FORM_DSC, %REF(3000), SIZE_CHECK );
    FDV$READ( %ASCID 'DEPOSIT', DPOSIT_FORM_DSC, %REF(2000), SIZE_DPOSIT );
```

```
!+
! Initialize account information
!-
INACCT();

!+
! Put up welcome form, wait for response
!-
FDV$CDISP( %ASCID 'WELCOME' );
SRVCHK();
FDV$WAIT();

!+
! Process all menu requests
!-
MENU();

!+
! Clean up and leave:
!   Close form library.
!   Reset keypad to numeric.
!   Delete a form from dynamic mem. res. form list just to show how.
!   Detach workspaces (not really necessary since DTERM would do it).
!   Detach terminal.
!-
FDV$LCLOS();
FDV$SPADA( %REF( 0 ) );
FDV$DEL( %ASCID 'MENU' );
FDV$DWKSP( WORKSPACE_DSC );
FDV$DWKSP( CHECKWKSP_DSC );
FDV$DTERM( TCA_DSC );
RETURN SS$_NORMAL;
END;
```

```
ROUTINE INACCT : NOVALUE =
!+
!       Read from file SAMP.DAT into internal variables.
!       Set up the workspace for checks and fill in the check form
!       with the account's name, address, and account number.
!-
    BEGIN

    OWN
        SAMP_FAB :      $FAB( FNM = 'FMS$EXAMPLES:SAMP.DAT', FAC = GET ),
        SAMP_RAB :      $RAB( FAB = SAMP_FAB ),
        STS;

    SAMP_RAB[RAB$W_USZ] = ACCOUNT_LEN;
    SAMP_RAB[RAB$L_UBF] = ACCOUNT_PTR;

    !+
    ! Open file, set account data
    !-
    IF NOT $OPEN( FAB = SAMP_FAB )
    THEN
        SIGNAL_STOP( .SAMP_FAB[FAB$L_STS], .SAMP_FAB[FAB$L_STV]);

    IF NOT $CONNECT( RAB = SAMP_RAB )
    THEN
        SIGNAL_STOP( .SAMP_RAB[RAB$L_STS], .SAMP_RAB[RAB$L_STV]);

    IF NOT $GET( RAB = SAMP_RAB )
    THEN
        SIGNAL_STOP( .SAMP_RAB[RAB$L_STS], .SAMP_RAB[RAB$L_STV]);

    !+
    ! Read the remaining records into the check register, counting them.
    ! The last record has the current balance, and some record has the
    ! last check number used (not necessarily the last record).
    !-
    LASTCHNUM = 0;
    LASTREGNUM = -1;
    SAMP_RAB[RAB$W_USZ] = REGITEM_LEN;
```

```
DO
    BEGIN
    SAMP_RAB[RAB$L_UBF] = REGARRAY[.LASTREGNUM+1, 0,0,0,0];

    IF NOT $GET(RAB=SAMP_RAB)
    THEN
        IF .SAMP_RAB[RAB$L_STS] EQL RMS$_EOF
        THEN
            EXITLOOP
        ELSE
            SIGNAL_STOP( .SAMP_RAB[RAB$L_STS], .SAMP_RAB[RAB$L_STV]);

    LASTREGNUM = .LASTREGNUM + 1;

    IF CH$NEQ( RI_NUM_LEN, REGARRAY[.LASTREGNUM,RI_NUM_FLD],
        4, UPLIT('    '), %C' ')
    THEN
        LASTCHNUM = VAL( LEN = RI_NUM_LEN, PTR = REGARRAY[.LASTREGNUM,RI_NUM_FLD] );
    END
WHILE .LASTREGNUM-1 LSS REGSIZE;

$CLOSE( FAB = SAMP_FAB );

!+
! Reach here as result of end of file (last record tried didn't read)
! or register filled and some records were not read.
! Check for no records or too many.
! Take balance from last record read.
! Set session sums to zero to say no activity yet.
!-
IF .LASTREGNUM EQL -1 OR .SAMP_RAB[RAB$L_STS] NEQ RMS$_EOF
THEN
    BEGIN
    LIB$PUT_OUTPUT( %ASCID 'DATA FILE IN ERROR' );
    $EXIT();
    END;
```

```
    BALANCE = VAL( LEN = RI_BALANCE_LEN,
                   PTR = REGARRAY[.LASTREGNUM,RI_BALANCE_FLD] );
    SBALANCE = .BALANCE;
    TOTDEP = 0;
    TOTPAY = 0;

    !+
    ! Set up the check workspace once so we don't have to do it every time.
    !-
    FMTCHK();

    END;



ROUTINE FMTCHK : NOVALUE =
!+
!       Format account data onto check form in the check workspace.
!-
    BEGIN
    LITERAL
        BUFFER_LEN =    MAX( FIRST_LEN+MIDDLE_LEN+LAST_LEN, CITY_LEN );

    LOCAL
        FIRST_LN :      WORD,
        MIDDLE_LN :     WORD,
        LAST_LN :       WORD,
        CITY_LN :       WORD,
        BUFFER :        VECTOR[ CH$ALLOCATION(BUFFER_LEN) ],
        BUFFER_DSC :    FIXDSC( BUFFER_LEN, 0 );

    BUFFER_DSC[DSC$A_POINTER] = CH$PTR(BUFFER);

    FDV$SWKSP( CHECKWKSP_DSC );
    FDV$LOAD( %ASCID 'CHECK' );
    STR$TRIM( FIRST_DSC, FIRST_DSC, FIRST_LN );
    STR$TRIM( MIDDLE_DSC, MIDDLE_DSC, MIDDLE_LN );
    STR$TRIM( LAST_DSC, LAST_DSC, LAST_LN );
```

```
CH$COPY( .FIRST_LN, ACCOUNT[FIRST_FLD],     1,      BLANK_PTR,
         1,         ACCOUNT[MIDDLE_FLD],    2,      PERIODBLANK_PTR,
         .LAST_LN,  ACCOUNT[LAST_FLD],
         %C' ',                             ! Fill character
         BUFFER_LEN, BUFFER );

FDV$PUT( BUFFER_DSC, %ASCID 'NAME' );
FDV$PUT( STREET_DSC, %ASCID 'STREET' );

STR$TRIM( CITY_DSC, CITY_DSC, CITY_LN );
CH$COPY( .CITY_LN,  ACCOUNT[CITY_FLD],      2,      COMMABLANK_PTR,
         STATE_LEN, ACCOUNT[STATE_FLD],     1,      BLANK_PTR,
         ZIP_LEN,   ACCOUNT[ZIP_FLD],
         %C' ',                             ! Fill character
         BUFFER_LEN, BUFFER );

FDV$PUT( BUFFER_DSC, %ASCID 'CSZ' );
FDV$PUT( HOMEPH_DSC, %ASCID 'HOMEPH' );
FDV$PUT( ACCTNO_DSC, %ASCID 'ACCTNO' );
FDV$SWKSP( WORKSPACE_DSC );
END;
```

```
ROUTINE MENU : NOVALUE =
'+
'       Accept inputs from the menu form and dispatch to the
!       appropriate routine.  Repeat until option 1 (exit) is
!       chosen.  The UARs in the form guarantee that we get back
!       only inputs '1'-'5' with the correct terminators.
!       Options are:
!               1 => Exit
!               2 => Write checks
!               3 => Make deposit
!               4 => View register
!               5 => View account data
'-
    BEGIN

    WHILE 1 DO
        BEGIN
        FDV$CDISP( %ASCID 'MENU' );
        SRVCHK();
        FDV$GET( OPTION_DSC, TERMINATOR, %ASCID 'OPTION' );
        CASE VAL( DSC = OPTION_DSC ) FROM 1 TO 5 OF
            SET
            [1]:
                ! Processing for EXIT menu choice.
                ! Do nothing but return.
                RETURN;
            [2]:
                WRITCH();
            [3]:
                MAKDEP();
            [4]:
                VUEREG();
            [5]:
                VUEACT();
            [OUTRANGE]:
                0;
            TES;
        END;
    END;
```

```
ROUTINE WRITCH : NOVALUE =
!+
!       Write one or more checks
!-
    BEGIN
    !+
    ! Turn on LED 3 on the VT100 during this routine, just to show how.
    !-
    FDV$LEDON( %REF( 3 ) );

    !+
    ! Mark WORKSPACE not displayed so it doesn't show up during refresh.
    ! Put up CHECK form from already loaded workspace
    ! and display current balance
    !-
    FDV$NDISP();
    FDV$SWKSP( CHECKWKSP_DSC );
    FDV$DISPW();

    FDV$PUT( STR$( .BALANCE ), %ASCID 'BALANCE' );
    !+
    ! Process checks until a keypad period is read
    !-
    TERMINATOR = 0;
    UNTIL .TERMINATOR EQL FDV$K_KP_PER DO
        BEGIN
        ONECHK();                               ! Process one check
        ENDCHK();                               ! Give options for continuing
        END;

    !+
    ! Turn off LED 3 on VT100
    !-
    FDV$LEDOF( %REF( 3 ) );
    FDV$SWKSP( WORKSPACE_DSC );

    RETURN
    END;
```

```
ROUTINE ONECHK : NOVALUE =
!+
! Process one check
!       If input is terminated by Kpd Period, return with no action
!       Else deduct from balance and enter into register.
!       Note that a UAR in the form guarantees that the amount of
!       the check is always less than or equal to the balance.
!       Note that the form function key UAR  allows only Kpd Period
!       as terminator (other than FDV$K_FT_NTR).
!-
    BEGIN
    FDV$PUT( STR$( .LASTCHNUM + 1 ), %ASCID 'NUMBER' );
    FDV$GETAL( JUNK_DSC, TERMINATOR );
    IF .TERMINATOR EQL FDV$K_KP_PER THEN RETURN;

    !+
    ! If the check wouldn't fit in the register, don't process, just
    ! give error message, wait for acknowledgement, and return
    !-
    IF .LASTREGNUM-1 EQL REGSIZE
    THEN
        BEGIN
        FDV$PUTL( %ASCID 'Register full, can''t enter check' );
        FDV$WAIT();
        RETURN
        END;

    !+
    ! Get amount from check.
    ! Update balance (in memory and on screen) and session sums.
    ! Transfer form values to register item.
    !-
    FDV$RET( RI_AMTPAY_DSC, %ASCID 'AMTPAY' );
    AMTPAY = VAL( DSC = RI_AMTPAY_DSC);
    BALANCE = .BALANCE - .AMTPAY;
    TOTPAY = .TOTPAY + .AMTPAY;

    FDV$PUT( STR$( .BALANCE ), %ASCID 'BALANCE' );
    FDV$RET( RI_BALANCE_DSC, %ASCID 'BALANCE');

    CH$FILL( %C' ', RI_AMTDEP_LEN, RI_AMTDEP_PTR );
    FDV$RET( RI_NUM_DSC, %ASCID 'NUMBER' );
```

```
    FDV$RET( RI_DATE_DSC, %ASCID 'DATE' );
    FDV$RET( RI_MEMPAYTO_DSC, %ASCID 'PAYTO' ); !Note: not from check's MEMO

    !+
    ! Update register array and counters
    ! (Note that the two step update (form->regitem->regarray)
    ! is done to make use of static descriptors).
    !-
    LASTREGNUM = .LASTREGNUM + 1;
    LASTCHNUM = .LASTCHNUM + 1;
    CH$MOVE( REGITEM_LEN, REGITEM, REGARRAY[.LASTREGNUM, 0,0,0,0]);

    RETURN
    END;



ROUTINE ENDCHK : NOVALUE =
!+
!       Finish off check processing by giving operator
!       three options:
!               RETURN  Write another check
!               KP0 0   Print the check into file SAMPCH.DAT
!               KP0 .   Return to menu
!       Check to see if check write was aborted by Kpd per.
!       If so, then don't give any further choice, just abort.
!       Note that form function key UAR allows only the above
!       terminators to get through.
!-
    BEGIN
    IF .TERMINATOR EQL FDV$K_KP_PER THEN RETURN;
    !+
    ! Tell the operator that the check has been paid by overlaying with
    ! a new form, using the normal workspace, thereby saving the check
    ! workspace in case another check is to be written.
    !-
    FDV$SWKSP( WORKSPACE_DSC );
    FDV$DISP( %ASCID 'CHECK_DONE' );
    SRVCHK();
```

```
!+
! Wait for operator to enter either KPD period, NTR, or KPD zero.
! Print the check as many times as requested.
! (Note that a UAR on the form guarantees that only those terminators
! are accepted).
! Process accordingly.
!-
FDV$WAIT( TERMINATOR );
WHILE .TERMINATOR EQL FDV$K_KP_0 DO
    BEGIN
    PRICHK();                          ! Print the check
    FDV$WAIT( TERMINATOR );
    END;
!+
! If choice is to quit
! then mark check wksp undisplayed so it doesn't appear during refresh.
! else mark normal workspace (occupied by CHECK_DONE form) undisplayed
!   so it doesn't show during refresh and then clear its lines.
!   (Clearing the space occupied by the CHECK_DONE form, lines 20-23
!   is better done by overlaying with a blank form to
!   avoid having to know the line numbers to clear).
!-
IF .TERMINATOR EQL FDV$K_KP_PER THEN
    BEGIN
    FDV$SWKSP( CHECKWKSP_DSC );
    FDV$NDISP();
    END
ELSE
    BEGIN
    FDV$NDISP();
    FDV$CLEAR( %REF( 20 ), %REF( 4 ) );
    FDV$SWKSP( CHECKWKSP_DSC );
    END;

!+
! Going to write another check now or eventually, so
! Clear out operator entered fields.
!-
FDV$PUTD( %ASCID 'AMTPAY' );
FDV$PUTD( %ASCID 'MEMO' );
FDV$PUTD( %ASCID 'PAYTO' );
RETURN
END;
```

```
ROUTINE PRICHK : NOVALUE =
!+
!       Print the check into the file SAMPCH.DAT
!       Use the check workspace, then switch back to the normal wksp
!       to keep things clean.
!-
    BEGIN
    OWN
        SAMPCH_FAB : $FAB( FAC = PUT, FNM = 'SAMPCH.DAT', RAT = CR, RFM = VAR),
        SAMPCH_RAB : $RAB( FAB = SAMPCH_FAB, RSZ = LINE_LEN, RBF = LINE_PTR );

    LOCAL
        LINE_LENGTH ;               Length of line image returned

    !+
    ! Open check writing file.  Note there's a new version for every check.
    ! Switch workspaces
    !-
    $CREATE( FAB = SAMPCH_FAB );
    $CONNECT( RAB = SAMPCH_RAB );
    FDV$SWKSP( CHECKWKSP_DSC );

    !+
      Get the top and bottom lines of the check from the named data
    ! (First two characters).
    !-
    FDV$RETDN( %ASCID 'FIRST', FIRSTL_DSC );
    SRVCHK();
    FDV$RETDN( %ASCID 'LAST', LASTL_DSC );
    SRVCHK();

    !+
    ! Get lines from form.
    ! Convert to line printer style.
    ! Write to file.
    !-
    INCR I FROM VAL( DSC = FIRSTL_DSC ) TO VAL( DSC = LASTL_DSC ) DO
        BEGIN
        FDV$RETFL( I, LINE_DSC, LINE_LENGTH, %REF(0) );
        $PUT( RAB = SAMPCH_RAB );
        END;
```

```
    FDV$PUTL( %ASCID 'Check written to file' );
    $CLOSE( FAB = SAMPCH_FAB );
    FDV$SWKSP( WORKSPACE_DSC );
    END;



ROUTINE MAKDEP : NOVALUE =
!+
!       Make a deposit, enter into check register
!       Cancel on Keypad period.
!       Note that the Form Function Key UAR allows only Kpd period.
!
! Put up deposit form with current balance
!-
    BEGIN
    LOCAL
        DEP;

    FDV$CDISP( %ASCID 'DEPOSIT' );
    SRVCHK();
    FDV$PUT( STR$( .BALANCE ), %ASCID 'CURBAL' );
    !+
    ! Get deposit amount and memo from operator.
    ! Abort on Kpd period.
    !-
    FDV$GETAL( DEPOSIT_DSC, TERMINATOR );
    IF .TERMINATOR EQL FDV$K_KP_PER THEN RETURN;
    !+
      Have deposit information now. If no room in check register
    ! must abort.
    !-
    IF ._ASTREGNUM-1 EQL REGSIZE
    THEN
        BEGIN
        FDV$PUTL( %ASCID 'Register full, can''t enter deposit' );
        FDV$WAIT();
        RETURN
        END;
```

```
!+
! Add to balance and session sum.
! Check for overflow (program and form keep only six digits).
! Display new balance.
! Make entry in register.
!-
DEP = VAL( DSC = DEP_AMT_DSC );
BALANCE = .BALANCE + .DEP;
TOTDEP = .TOTDEP + .DEP;
IF .BALANCE GEQ 1000000
THEN
    BEGIN
    BALANCE = .BALANCE - 1000000;
    FDV$PUTL( %ASCID %STRING('Overflow in bank computer, only 6 digits ',
                        'allowed, we keep the rest of the money') );
    FDV$WAIT();
    END;

FDV$PUT( STR$( .BALANCE ), %ASCID 'NEWBAL' );

CH$FILL( %C' ', RI_NUM_LEN, REGITEM[RI_NUM_FLD] );  ! Blank since it's not a check

CH$COPY( DEP_DATE_LEN, DEP_DATE_PTR, %C' ', RI_DATE_LEN, RI_DATE_PTR );

CH$COPY( DEP_MEMO_LEN, DEP_MEMO_PTR, %C' ', RI_MEMPAYTO_LEN, RI_MEMPAYTO_PTR );

CH$COPY( DEP_AMT_LEN, DEP_AMT_PTR, %C' ', RI_AMTDEP_LEN, RI_AMTDEP_PTR );

CH$FILL( %C' ', RI_AMTPAY_LEN, RI_AMTPAY_PTR );     ! Blank since it's not a check

FDV$RET( RI_BALANCE_DSC, %ASCID 'NEWBAL' ); ! Avoids need to format RI_BALANCE$

LASTREGNUM = .LASTREGNUM + 1;

CH$MOVE( REGITEM_LEN, REGITEM, REGARRAY[.LASTREGNUM, 0,0,0,0]);

!+
! Sample of how to keep message texts stored with the form rather
! than in a program.  This is especially useful for multi-lingual
! environments: only the form text and the form named data must
! be changed and nothing in the program.  The trick is to store the
! response text in named data. This is the only example of how to do
```

```
! it in this program, but all messages could be stored like this.
! Message intent is: "Deposit made, press RETURN or ENTER to continue."
!-
FDV$RETDN( %ASCID 'DONE', DONE_DSC );
FDV$PUTL( DONE_DSC );
FDV$WAIT();
RETURN
END;
```

```
ROUTINE VUEREG : NOVALUE =
!+
!       View the check register and scroll through it.
!       Also display totals for current session.
!
! Put up register form.
! Check for current session totals overflow. If so, output 'OVRFLO'
! Put out summary of this session into indexed(4) fields.
!-
    BEGIN
    LOCAL
        DEPDSP,
        PAYDSP;

    FDV$CDISP( %ASCID 'REGISTER' );
    SRVCHK();
    IF .TOTDEP LSS 1000000
    THEN
        DEPDSP = STR$( .TOTDEP )
    ELSE
        DEPDSP = %ASCID 'OVRFLO';
    IF .TOTPAY LSS 1000000
    THEN
        PAYDSP = STR$( .TOTPAY )
    ELSE
        PAYDSP = %ASCID 'OVRFLO';
    FDV$PUT( STR$( .SBALANCE ), %ASCID 'SUMMARY', %REF( 1 ) );
    FDV$PUT( .DEPDSP, %ASCID 'SUMMARY', %REF( 2 ) );
    FDV$PUT( .PAYDSP, %ASCID 'SUMMARY', %REF( 3 ) );
    FDV$PUT( STR$( .BALANCE ), %ASCID 'SUMMARY', %REF( 4 ) );
    !+
    ! Get number of lines in scroll area from form named data (item 1).
    !-
    FDV$RETDI( %REF( 1 ), NSCROLL_DSC );
    SRVCHK();
    NSCROL = VAL( DSC = NSCROLL_DSC );
    !+
    ! Put lines from check register array into scrolled area.
    ! The window is initially from item 1 up to item
    ! min(NSCROL,LASTREGNUM), that is, up to the size of the scrolled
    ! area or the size of the register, whichever is less. Assume there
    ! is at least one line (the initial deposit).
    !-
```

```
MINWINDOW = 1;
FDV$PUTSC( %ASCID 'NUMBER', REGARRAY_DSC[0] );      ! First line
CURLINE = 1;                                        ! Res item cursor is on
WHILE ( .CURLINE-1 LSS .LASTREGNUM AND .CURLINE LSS .NSCROL ) DO
    BEGIN
    CURLINE = .CURLINE + 1;
    FDV$PFT( %REF(FDV$K_FT_SFW), %ASCID 'NUMBER' );
    FDV$PUTSC( %ASCID 'NUMBER', REGARRAY_DSC[ .CURLINE-1 ] );
    END;
MAXWINDOW = .CURLINE;
!+
! Get input from fake field of scrolled line and do what it says:
!   Kpd . or RETURN/ENTER   => return to menu
!   UPARROW or TAB          => scroll forward
!   DOWNARROW or BACKSPACE  => scroll backward
!   all others              => ignore
! Note that there is no form function key UAR so this routine
! handles all terminators itself (by ignoring illegal ones).
!-
WHILE 1 DO
    BEGIN
    FDV$GET( FAKE_DSC, TERMINATOR, %ASCID 'FAKE' );
    SELECTONE .TERMINATOR OF
        SET
        [ FDV$K_FT_NTR, FDV$K_KP_PER ]:     RETURN;
        [ FDV$K_FT_SFW, FDV$K_FT_SNX ]:     SCRFWD();
        [ FDV$K_FT_SBK, FDV$K_FT_SPR ]:     SCRBAK();
        TES;
    END;
END;
```

```
ROUTINE SCRFWD : NOVALUE =
!+
! Scroll forward.
!       CURLINE is the line in the register that the cursor is on.
!       MINWINDOW and MAXWINDOW delimit the part of the register
!       currently displayed in the scrolled area
!-
    BEGIN
    !+
    ! If cursor is at the end of the register, report, and return
    !-
    IF .CURLINE-1 EQL .LASTREGNUM
    THEN
        BEGIN
        FDV$PUTL( %ASCID 'Last line of register' );
        RETURN
        END;
    !+
    ! If cursor not at the last line of a window, just move down
    ! If cursor is  at the last line of a window,
    !   move window forward one line,
    !   write the new last line to the last line of the scrolled area
    ! Move current line pointer forward
    !-
    IF .CURLINE NEQ .MAXWINDOW
    THEN
        FDV$PFT( %REF(FDV$K_FT_SFW), %ASCID 'NUMBER' )
    ELSE
        BEGIN
        MINWINDOW = .MINWINDOW + 1;
        MAXWINDOW = .MAXWINDOW + 1;
        FDV$PFT( %REF(FDV$K_FT_SFW), %ASCID 'NUMBER', REGARRAY_DSC[ .MAXWINDOW - 1 ] );
        END;
    CURLINE = .CURLINE + 1;
    RETURN
    END;
```

```
ROUTINE SCRBAK : NOVALUE =
!+
! Scroll backward.
!       CURLINE is the line in the register that the cursor is on.
!       MINWINDOW and MAXWINDOW delimit the part of the register
!       currently displayed in the scrolled area
!-
    BEGIN
    !+
    ! If the cursor is at the beginning of the register, report, and return
    !-
    IF .CURLINE EQL 1
    THEN
        BEGIN
        FDV$PUTL( %ASCID 'First line of register' );
        RETURN
        END;
    !+
    ! If cursor not at first line of the window, just move up
    ! If cursor is  at first line of the window,
    !   move window back one line,
    !   write the new first line to the first line of the scrolled area
    ! Move current line pointer back
    !-
    IF .CURLINE NEQ .MINWINDOW
    THEN
        FDV$PFT( %REF(FDV$K_FT_SBK), %ASCID 'NUMBER' )
    ELSE
        BEGIN
        MINWINDOW = .MINWINDOW - 1;
        MAXWINDOW = .MAXWINDOW - 1;
        FDV$PFT( %REF(FDV$K_FT_SBK), %ASCID 'NUMBER', REGARRAY_DSC[ .MINWINDOW - 1 ] );
        END;
    CURLINE = .CURLINE - 1;
    RETURN
    END;
```

```
ROUTINE VUEACT : NOVALUE =
!+
!       View the account data.
!       If operator knows the secret word, let operator change
!       the account data for this session.
!-
    BEGIN
    FDV$CDISP( %ASCID 'ACCOUNT_DATA' );
    SRVCHK();
    FDV$PUTAL( ACCOUNT_DSC );
    FDV$PUTD( %ASCID 'SECRET');
    !+
    ! This is not the best way to do protection, just a way of showing
    ! another FMS feature. At this point, supervisor mode is on, so the
    ! only input allowed is to the password field.
    ! If operator doesn't know password, return to menu.
    !-
    FDV$GETAL( 0, TERMINATOR ); ! Don't care about value now
    IF .TERMINATOR EQL FDV$K_KP_PER THEN RETURN;
    FDV$RET( PASSWORD_DSC, %ASCID 'SECRET' );
    IF CH$NEQ( OPW_LEN, ACCOUNT[OPW_FLD], PASSWORD_LEN, PASSWORD_PTR, %C' ')
    THEN
        RETURN;
    !+
      Allow input from other fields and read from them.
    ! If read is terminated by keypad period, don't change account.
    !-
    FDV$SPOFF();
    GETAL();                                ! Read all fields
    FDV$SPON();                             ! Not really needed, just showing off.
    IF .TERMINATOR NEQ FDV$K_KP_PER
    THEN
        BEGIN
        FDV$RETAL( ACCOUNT_DSC );
        FMTCHK();                           ! Update the check workspace
        END;
    RETURN
    END;
```

```
ROUTINE GETAL : NOVALUE =
!+
! Simulate action of FDV$GETAL, using FDV$GETAF and PFT.  Could
! replace this whole routine with a call on FDV$GETAL, but this shows
! how mainline program can allow same operator freedom of filling in
! fields but still regain control after each or changed field.
! Technique is to read any field, looking only at terminator, then do
! a process field terminator call to do the operator's action.
! This technique can be used with calls on FDV$GET or FDV$GETAF.
  This example starts with a GET on field '*', first field on form.
!-
    BEGIN
    LOCAL
        FIELDINDEX;

    FDV$GET( JUNK_DSC, TERMINATOR, %ASCID '*' );
    FDV$RETFN( FIELDNAME_DSC, FIELDINDEX );      !Get first field's name
    WHILE 1 DO
        BEGIN
        !+
        ! Do any special processing for field FIELDNAME at this point.
        ! ...
        ! Go to next or previous field or leave form
        !-
        FDV$PFT( TERMINATOR );
        !+
        ! If status is error, then PFT failed because terminator was
        ! a Keypad Key, which means return to caller.
        !-
        IF .FMSSTATUS LSS 0 THEN RETURN;
        IF .TERMINATOR EQL FDV$K_PT_NTR
        THEN
            IF .FMSSTATUS NEQ 2          ! Form incomplete
            THEN
                RETURN
            ELSE
                BEGIN
                FDV$PUTL( %ASCID 'INPUT REQUIRED' );
                FDV$BELL();
                END;
```

```
        .+
        ! Go set any other field, returning its name
        !-
        FDV$GETAF( JUNK_DSC, TERMINATOR, FIELDNAME_DSC, FIELDINDEX );
        END;
    RETURN
    END;



ROUTINE GETSTA : NOVALUE =
!+
! Check FMS status by calling FDV$STAT.
! If not success (>0), print and stop
!-
    BEGIN
    FDV$STAT( FMSSTATUS, RMSSTATUS );
    IF .FMSSTATUS GTR 0 THEN RETURN;
    ABORT();                ! and never come back
    END;



ROUTINE SRVCHK : NOVALUE =
!+
! Check FMS status by looking at the status recording variables.
!-
    BEGIN
    IF .FMSSTATUS GTR 0 THEN RETURN;
    ABORT();                ! and never come back
    END;
```

```
ROUTINE ABORT : NOVALUE =
!+
! There is an error returned in the status variables. Detach the
! terminal to clean up, then print the errors, and stop.
!-
    BEGIN
    LOCAL
        BUFFER : VECTOR[CH$ALLOCATION(80)],
        BUFFER_DSC :    FIXDSC(80,0);

    BUFFER_DSC[DSC$A_POINTER] = CH$PTR(BUFFER);

    FDV$DTERM( TCA_DSC );
    LIB$SYS_FAO( %ASCID 'FDV ERROR.!/!_FMS STATUS:!SL!/RMSSTATUS:!SL', 0,
        BUFFER_DSC,
        .FMSSTATUS, .RMSSTATUS );
    LIB$PUT_OUTPUT( BUFFER_DSC );
    $EXIT(CODE=.FMSSTATUS);
    END;
```

```
GLOBAL ROUTINE VALID1 =
!+
! VALID1
!       UAR for field validation of any one character field. The
!       UAR associated data has in it the legal characters allowed,
!       except that blank is not allowed unless it appears before
!       the first trailing blank.  For example an assoc. value string
!       'aqr' implies that only the letters a, q, and r are allowed.
!       A string ' aqr' means that blank is acceptable in addition
!       to a, q, and r. Note that this routine is case sensitive
!       (that is, it checks for correct case).  You can get around
!       case sensitivity by using the force upper case field attribute
!       and putting only capitals into the UAR associated value
!       string.
!
!       This routine can be used with any form and field since
!       it determines the context for itself.
!-
    BEGIN
    LOCAL
        TCA,
        WKSP,
        CURPOS,
        FLDTRM,
        FINDEX;

    FIXSTR(
        FRMNAM, 31,
        UARVAL, 80,
        FLDNAME, 31,
        FVALUE, 1 );

    !+
    ! Retrieve context; we will ignore TCA address, WKSP address, FRMNAM,
    !   CURPOS, FLDTRM, and INSOVR, using only UARVAL, and only the
    !   initial, non-blank characters of it.
    ! Retrieve field name and index.
    ! Retrieve field value.
    FDV$RETCX( TCA, WKSP, FRMNAM_DSC, UARVAL_DSC, CURPOS, FLDTRM, INSOVR, HELPNUM );
    FDV$RETFN( FLDNAME_DSC, FINDEX );
    FDV$RET( FVALUE_DSC, FLDNAME_DSC, FINDEX );
```

```
    !+
    ! To be valid, FVALUE must occur in the string UARVAL
    !-
    IF CH$FAIL(CH$FIND_SUB(UARVAL_LEN, UARVAL_PTR, FVALUE_LEN, FVALUE_PTR ))
    THEN
        BEGIN
        FDV$PUTL( %ASCID 'Illegal value' );
        RETURN FDV$K_UVAL_FAIL
        END
    ELSE
        RETURN FDV$K_UVAL_SUC                    !Success
    END;


GLOBAL ROUTINE TAKE15 =
!+
! Function Key User Action Routine for the MENU form of SAMP.
! Convert Keypad 1-5 into field values 1-5.
! Convert Keypad Period into field value 1.
! Reject all other function keys with error message.
!-
    BEGIN
    LOCAL
        TCA,
        WKSP,
        CURPOS,
        FLDTRM,
        VALUE;

    FIXSTR(
        FRMNAM, 4,
        UARVAL, 1 );

    !+
    ! Retrieve context; we will ignore TCA address, WKSP address, FRMNAM,
    !   UARVAL, CURPOS and INSOVR, using only FLDTRM
    FDV$RETCX( TCA, WKSP, FRMNAM_DSC, UARVAL_DSC, CURPOS, FLDTRM, INSOVR, HELPNUM );
```

```
    !+
    ! Do the conversion, displaying the value converted if found.
    ! Reject if not one of the expected terminators.
    !
    SELECTONE .FLDTRM OF
        SET
        [FDV$K_KP_1,FDV$K_KP_PER]:      VALUE = %ASCID '1';
        [FDV$K_KP_2]:                   VALUE = %ASCID '2';
        [FDV$K_KP_3]:                   VALUE = %ASCID '3';
        [FDV$K_KP_4]:                   VALUE = %ASCID '4';
        [FDV$K_KP_5]:                   VALUE = %ASCID '5';
        [OTHERWISE]:
            BEGIN
            FDV$PUTL( %ASCID 'Illegal function key' );
            FDV$SIGOP();
            RETURN FDV$K_UKEY_SUC;          ! Just ignore it now
            END;
        TES;
    FDV$PUT( .VALUE, %ASCID 'OPTION' );
    ! Treat as if it is RETURN
    RETURN FDV$K_UKEY_NTR
    END;



GLOBAL ROUTINE PASSKY =
!+
! General function key var to pass only those from the (small) list
! in the var associated value string and reject all others.
! The list is of the form: n <oneblank> n <oneblank> ... n <manyblanks>
! For example the string '110 112' would accept Keypad period and
! Keypad zero but no other function keys.
!-
    BEGIN
    LOCAL
        TCA,
        WKSP,
        CURPOS,
        FLDTRM,
        REM_LEN,
        NONBLANK,
        NEXTBLANK;
```

```
    FIXSTR(
        FRMNAM, 4,
        UARVAL, 82 );

    !+
    ! Retrieve context: we will ignore TCA address, WKSP address, FRMNAM,
    !    INSOVR, and CURPOS, using only FLDTRM and UARVAL.
    !-
    FDV$RETCX( TCA, WKSP, FRMNAM_DSC, UARVAL_DSC, CURPOS, FLDTRM, INSOVR, HELPNUM );

    !+
    ! Break up the list into numbers.  Check each against the actual
    ! terminator. If terminator found in list, return success.
    !-
    REM_LEN = UARVAL_LEN;
    NONBLANK = UARVAL_PTR;                  ! Beginning of string
    WHILE CH$RCHAR( .NONBLANK ) NEQ %C' ' AND .REM_LEN GTR 0 DO
        BEGIN
        NEXTBLANK = CH$FIND_CH( .REM_LEN, .NONBLANK, %C' ' );
        IF CH$FAIL(.NEXTBLANK) THEN EXITLOOP;
        IF .FLDTRM EQL VAL( DSC = SEG$( .NONBLANK, .NEXTBLANK ) )
        THEN
            RETURN FDV$K_UKEY_TRM;          !Pass key to application
        NONBLANK = CH$PLUS(.NEXTBLANK, 1);
        REM_LEN = UARVAL_LEN - CH$DIFF( .NONBLANK, UARVAL_PTR );
        END;
    RETURN FDV$K_UKEY_ERR                   !Let FDV do the beeping
    END;



GLOBAL ROUTINE CHKCHK =
!+
! UAR for SAMP CHECK form.  Makes sure that the check amount is
! less than or equal to the current balance.  If not, complain and
! change video attributes on balance field so the potential bouncer
! can see what there is to work with.
!-
    BEGIN
    LOCAL
        BLINKBOLD;
```

```
    FIXSTR(
        BALANCE, 6,
        AMTPAY, 6 );

    FDV$RET( BALANCE_DSC, %ASCID 'BALANCE' );
    FDV$RET( AMTPAY_DSC, %ASCID 'AMTPAY' );

    IF VAL( DSC = BALANCE_DSC ) GEQ VAL( DSC = AMTPAY_DSC )
    THEN
        BEGIN
        BLINKBOLD = -1;                     !Restore to original
        FDV$APVA( BLINKBOLD, %ASCID 'BALANCE' );
        RETURN FDV$K_UVAL_SUC;
        END
    ELSE
        BEGIN
        BLINKBOLD = 3;                      !Make it very visible
        FDV$APVA( BLINKBOLD, %ASCID 'BALANCE' );
        FDV$PUTL(
            %ASCID 'Your balance doesn''t cover that much, reenter amount' );
        RETURN FDV$K_UVAL_FAIL;
        END;
    END;



GLOBAL ROUTINE RANGE =
!+
! General purpose UAR to check the range of any numeric item.  The
! associated UAR data must have one of the four forms:
!       L,U,space,{message}
!       ,U,space,{message}
!       L,,space,{message}
!       ,,space,{message}
! where L is lower bound, U is upper bound, and {message} is an
! optional error message in case the field value is out of bounds.
! If one of the bounds isn't given, it isn't checked for.  If neither
! bound is given, nothing is checked, everything succeeds.  If the
! UAR value doesn't have a comma, a FDV$_UAR error message is returned
! to the calling program by the FDV so the form designer has to go
! back and do it right.  If no {message} is given, a simple
! "out of range U:L" message is given to the hapless operator.
```

```
! This UAR can work with any form and numeric field since it sets
! context itself.  Care must be taken with fields using field marker
! periods since those periods are not returned to the program.
!-
    BEGIN
    LOCAL
        TCA,
        WKSP,
        CURPOS,
        FLDTRM,
        BLANK,
        COMMA,
        NUMBER,
        INDEX,
        LENGTH;

    FIXSTR(
        FRMNAM, 31,
        UARVAL, 80,
        NAME, 31,
        NUMBER, 132 );

    LABEL
        CHECK_BOUNDS;

    !+
    ! Get context which yields associated data value (ignore other stuff).
    ! Get current field name and index.
    ! Get field value.
    !-
    FDV$RETCX( TCA, WKSP, FRMNAM_DSC, UARVAL_DSC, CURPOS, FLDTRM, INSOVR, HELPNUM );
    FDV$RETFN( NAME_DSC, INDEX );
    FDV$RET( NUMBER_DSC, NAME_DSC, INDEX );
    NUMBER = VAL( DSC = NUMBER_DSC );
```

```
!+
! Find comma and blank delimiters.
!-
IF CH$FAIL(COMMA = CH$FIND_CH( UARVAL_LEN, UARVAL_PTR, %C',' ) )
THEN
    RETURN 0;        ! Illegal UARVAL string, FDV returns error

LENGTH = CH$DIFF(.COMMA, UARVAL_PTR);
BLANK = CH$FIND_CH( UARVAL_LEN - .LENGTH, .COMMA, %C' ' );

CHECK_BOUNDS:
    BEGIN
    !+
    ! Check for lower bound.
    !-
    IF .LENGTH NEQ 0
    THEN
        IF .NUMBER LSS VAL( LEN = .LENGTH, PTR = UARVAL_PTR )
        THEN
            LEAVE CHECK_BOUNDS;

    !+
    ! Check for upper bound
    !-
    LENGTH = CH$DIFF(.BLANK, .COMMA + 1);
    IF .LENGTH NEQ 0
    THEN
        IF .NUMBER GTR VAL( LEN = .LENGTH, PTR = .COMMA + 1 )
        THEN
            LEAVE CHECK_BOUNDS;

    !+
    ! Passed both tests successfully, return success for UAR value
    !-
    RETURN FDV$K_UVAL_SUC;
    END;             ! CHECK_BOUNDS
```

```
    !+
    ! Error in one of the bounds.
    ! Give error message: either from the UARVAL or make one up.
    !-
    IF CH$RCHAR( .BLANK + 1 ) NEQ %C' '
    THEN
        FDV$PUTL( SEG$( .BLANK + 1, UARVAL_PTR + UARVAL_LEN ) )
    ELSE
        BEGIN
        OWN TMP_DSC : DYNDSC;
        LIB$SYS_FAC( %ASCID'Field value out of bounds. Must be in range "!AS".',
            0, TMP_DSC, SEG$( UARVAL_PTR, .BLANK ) );
        FDV$PUTL( TMP_DSC );
        END;
    FDV$SIGOP();                    !Beep, too.
    RETURN FDV$K_UVAL_FAIL
    END;
END
ELUDOM
```

```
!+
!  FDVDEF.REQ
!
!  BLISS Interface require file for VAX-11 FMS V2
!-
!+
!  External declarations for all form driver routines follow.
!-
EXTERNAL ROUTINE

      FDV$ADLVA
     ,FDV$AFCX
     ,FDV$AFVA
     ,FDV$ATERM
     ,FDV$AWKSP
     ,FDV$BELL
     ,FDV$CANCL
     ,FDV$CDISP
     ,FDV$CLEAR
     ,FDV$CLRSH
     ,FDV$DPCOM
     ,FDV$DEL
     ,FDV$DFKBD
     ,FDV$DISP
     ,FDV$DISPW
     ,FDV$DTERM
     ,FDV$DWKSP
     ,FDV$GET
     ,FDV$GETAF
     ,FDV$GETAL
     ,FDV$GETDL
     ,FDV$GETSC
     ,FDV$ILTRM
     ,FDV$LCHAN
     ,FDV$LCLOS
     ,FDV$LEDOF
     ,FDV$LEDON
     ,FDV$LOAD
     ,FDV$LOPEN
     ,FDV$NDISP
     ,FDV$PFT
     ,FDV$PLT
```

```
      ,FDV$PUTAL
      ,FDV$PUTD
      ,FDV$PUTDA
      ,FDV$PUTL
      ,FDV$PUTSC
      ,FDV$READ
      ,FDV$RET
      ,FDV$RETAL
      ,FDV$RETCX
      ,FDV$RETDI
      ,FDV$RETDN
      ,FDV$RETFL
      ,FDV$RETFN
      ,FDV$RETFO
      ,FDV$RETLE
      ,FDV$RFRSH
      ,FDV$SHOW
      ,FDV$SIGOP
      ,FDV$SPADA
      ,FDV$SPOFF
      ,FDV$SPON
      ,FDV$SSIGQ
      ,FDV$SSRV
      ,FDV$STAT
      ,FDV$STERM
      ,FDV$STIME
      ,FDV$SWKSP
      ,FDV$TCHAN
      ,FDV$WAIT
      ;

!*****************************************************************************
! FDV Status code definitions.
! First the definitions for the codes returned by FDV$STAT. The system
! independent status codes.
!*****************************************************************************

LITERAL                     ! Success codes.

    FDV$K_SUC = 1,          !   Success.
    FDV$K_INC = 2,          !   Form incomplete.
    FDV$K_MOD = 3;          !   Field modified.
```

```
LITERAL                         ! Failure codes.

    FDV$K_IMP = -2,             !   Workspace too small.
    FDV$K_FSP = -3,             !   File spec. invalid.
    FDV$K_IOL = -4,             !   Error openins FLB.
    FDV$K_FLB = -5,             !   File is not a FLB.
    FDV$K_ICH = -6,             !   Invalid channel number.
    FDV$K_FCH = -7,             !   FLB not open on channel.
    FDV$K_FRM = -8,             !   Invalid BFD.
    FDV$K_FNM = -9,             !   Form not in FLB.
    FDV$K_LIN = -10,            !   Bad line or offset.
    FDV$K_FLD = -11,            !   Field not in form.
    FDV$K_NOF = -12,            !   No fields in form.
    FDV$K_DSP = -13,            !   Display only field.
    FDV$K_NSC = -14,            !   Field not scrolled.
    FDV$K_DNM = -15,            !   Named data not found.
    FDV$K_DLN = -16,            !   Data is too lons for output field.
    FDV$K_UTR = -17,            !   Undefined field terminator.
    FDV$K_IOR = -18,            !   Error readins FLB.
    FDV$K_IFN = -19,            !   Invalid context for PFT call.
    FDV$K_ARG = -20,            !   Wrons number of arsuments.
    FDV$K_INI = -21,            !   Workspace not initialized.
    FDV$K_STR = -22,            !   Strins too larse.
    FDV$K_FVM = -23,            !   Error freeins virtual memory.
    FDV$K_IVM = -24,            !   Insufficient virtual memory.
    FDV$K_ITT = -25,            !   Invalid terminal type.
    FDV$K_TCA = -26,            !   TCA invalid or undefined.
    FDV$K_STA = -27,            !   TCA too small.
    FDV$K_WID = -28,            !   Form too wide for terminal or ctx.
    FDV$K_NFL = -29,            !   No form loaded in current wksp.
    FDV$K_IBF = -30,            !   User buffer to small for form read.
    FDV$K_NDS = -31,            !   Form not displayed tried to do set
    FDV$K_UDP = -33,            !   UAR depth limit reached
    FDV$K_UAR = -34,            !   UAR returned illesal status
    FDV$K_UNF = -35,            !   UAR specified in form, not found in vector
    FDV$K_CAN = -39,            !   Cancelled directive
    FDV$K_KIF = -40,            !   Illesal Keyfunction
    FDV$K_KEX = -41,            !   Too many Keycode for Keyfunction
    FDV$K_KTW = -42,            !   Two Keyfunctions for a Keycode
    FDV$K_KIL = -43,            !   Keycode illesal
    FDV$K_TMO = -44,            !   Timeout on set...
    FDV$K_LLI = -45,            !   Lensth of Line Imase too lons for FDV
```

```
    FDV$K_VAL = -47,     !   Parameter value out of range
    FDV$K_IFU = -48,     !   Illegal function in UAR
    FDV$K_SYS = -49;     !   System error during terminal I/O

!++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
! FDV status codes returned when FDV$... routines are called as functions.
! These codes are VMS status codes and can be signalled.  They correspond
! one-to-one with the FMS status codes retrievable from FDV$STAT.
!++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
LITERAL
    FDV$_SUC = 2719889,
    FDV$_INC = 2719897,
    FDV$_MOD = 2719905,
    FDV$_IMP = 2719922,
    FDV$_FSP = 2719930,
    FDV$_IOL = 2719938,
    FDV$_FLB = 2719946,
    FDV$_ICH = 2719954,
    FDV$_FCH = 2719962,
    FDV$_FRM = 2719970,
    FDV$_FNM = 2719978,
    FDV$_LIN = 2719986,
    FDV$_FLD = 2719994,
    FDV$_NOF = 2720002,
    FDV$_DSP = 2720010,
    FDV$_NSC = 2720018,
    FDV$_DNM = 2720026,
    FDV$_DLN = 2720034,
    FDV$_UTR = 2720042,
    FDV$_IOR = 2720050,
    FDV$_IFN = 2720058,
    FDV$_ARG = 2720066,
    FDV$_INI = 2720074,
    FDV$_STR = 2720082,
    FDV$_IVM = 2720090,
    FDV$_FVM = 2720098,
    FDV$_ITT = 2720106,
    FDV$_TCA = 2720114,
    FDV$_STA = 2720122,
    FDV$_WID = 2720130,
    FDV$_NFL = 2720138,
    FDV$_IBF = 2720146,
```

```
    FDV$_NDS = 2720154,
    FDV$_UDP = 2720162,
    FDV$_UAR = 2720170,
    FDV$_UNF = 2720178,
    FDV$_CAN = 2720194,
    FDV$_KIF = 2720202,
    FDV$_KEX = 2720210,
    FDV$_KTW = 2720218,
    FDV$_KIL = 2720226,
    FDV$_TMO = 2720234,
    FDV$_LLI = 2720242,
    FDV$_VAL = 2720250,
    FDV$_IFU = 2720258,
    FDV$_SYS = 2720266;

!************************
! FMS terminator codes:
!************************
LITERAL  FDV$K_FT_NTR     =  0  !Enter (i.e. end GETs)
        ,FDV$K_FT_NXT     =  1  !Next field
        ,FDV$K_FT_PRV     =  2  !Previous field
        ,FDV$K_FT_ATB     =  3  !Automatically move to next field
        ,FDV$K_FT_XBK     =  4  !Exit scrolled area backward
        ,FDV$K_FT_XFW     =  5  !Exit scrolled area forward
        ,FDV$K_FT_SNX     =  6  !Scroll forward to next field
        ,FDV$K_FT_SPR     =  7  !Scroll backward to previous field
        ,FDV$K_FT_SFW     =  8  !Scroll forward
        ,FDV$K_FT_SBK     =  9  !Scroll backward
        ,FDV$K_FT_ILG_NXT = 11  !Illegal context for next field
        ,FDV$K_FT_ILG_PRV = 12  !Illegal context for previous field
        ,FDV$K_FT_ILG_ATB = 13  !Illegal context for auto move to next field
        ,FDV$K_FT_ILG_XBK = 14  !Illegal context for exit scrolled area backward

        ,FDV$K_FT_ILG_XFW = 15  !Illegal context for exit scrolled area forward
        ,FDV$K_FT_ILG_SFW = 16  !Illegal context for scroll forward
        ,FDV$K_FT_ILG_SBK = 17; !Illegal context for scroll backward

!************************************************************
! Function key terminators returned from GETs and WAIT
! Also used as FDV keycodes for use with DFKBD.
!************************************************************
```

```
LITERAL  FDV$K_AR_UP      =  99,
        ,FDV$K_AR_DOWN    =  100,
        ,FDV$K_AR_LEFT    =  101,
        ,FDV$K_AR_RIGHT   =  102,
        ,FDV$K_PF_1       =  103,
        ,FDV$K_PF_2       =  104,
        ,FDV$K_PF_3       =  105,
        ,FDV$K_PF_4       =  106,
        ,FDV$K_KP_NTR     =  107,
        ,FDV$K_KP_COM     =  108,
        ,FDV$K_KP_HYP     =  109,
        ,FDV$K_KP_PER     =  110,
        ,FDV$K_KP_0       =  112,
        ,FDV$K_KP_1       =  113,
        ,FDV$K_KP_2       =  114,
        ,FDV$K_KP_3       =  115,
        ,FDV$K_KP_4       =  116,
        ,FDV$K_KP_5       =  117,
        ,FDV$K_KP_6       =  118,
        ,FDV$K_KP_7       =  119,
        ,FDV$K_KP_8       =  120,
        ,FDV$K_KP_9       =  121,
        ,FDV$K_GAR_UP     =  227,
        ,FDV$K_GAR_DOWN   =  228,
        ,FDV$K_GAR_RIGHT  =  229,
        ,FDV$K_GAR_LEFT   =  230,
        ,FDV$K_GPF_1      =  231,
        ,FDV$K_GPF_2      =  232,
        ,FDV$K_GPF_3      =  233,
        ,FDV$K_GPF_4      =  234,
        ,FDV$K_GKP_NTR    =  235,
        ,FDV$K_GKP_COM    =  236,
        ,FDV$K_GKP_HYP    =  237,
        ,FDV$K_GKP_PER    =  238,
        ,FDV$K_GKP_0      =  240,
        ,FDV$K_GKP_1      =  241,
        ,FDV$K_GKP_2      =  242,
        ,FDV$K_GKP_3      =  243,
        ,FDV$K_GKP_4      =  244,
        ,FDV$K_GKP_5      =  245,
        ,FDV$K_GKP_6      =  246,
        ,FDV$K_GKP_7      =  247
```

```
        ,FDV$K_GKP_8      =  248
        ,FDV$K_GKP_9      =  249 ;
!*****************************************
! FDV KeyFunctions. For use in DFKBD call.
!*****************************************
literal  FDV$K_KF_GOLD   =   1
        ,FDV$K_KF_RESET  =   2
        ,FDV$K_KF_CRSLF  =   3
        ,FDV$K_KF_CRSRT  =   4
        ,FDV$K_KF_DLCHR  =   5
        ,FDV$K_KF_DLFLD  =   6
        ,FDV$K_KF_INS    =   7
        ,FDV$K_KF_OVR    =   8
        ,FDV$K_KF_RFRSH  =   9
        ,FDV$K_KF_HELP   =  10
        ,FDV$K_KF_NXT    =  11
        ,FDV$K_KF_PRV    =  12
        ,FDV$K_KF_NTR    =  13
        ,FDV$K_KF_SBK    =  14
        ,FDV$K_KF_SFW    =  15
        ,FDV$K_KF_XBK    =  16
        ,FDV$K_KF_XFW    =  17
        ,FDV$K_KF_NONE   =   0
        ,FDV$K_KF_DFLT   =  -1 ;


!*****************************************************************
! UAR return codes. These codes are returned by UAR to FDV.
!*****************************************************************
! Field completion return codes
!*****************************
LITERAL  FDV$K_UVAL_SUC =  1000 !Field completion success
        ,FDV$K_UVAL_FAIL=  1001 !Field completion failure
        ,FDV$K_UVAL_END =  1002 !Field completion suc-stop UARs
!************************
! Help UAR return codes
!************************
        ,FDV$K_UHELP_NO =  2000 !No help given, try next step
        ,FDV$K_UHELPED  =  2001 !Help given, continue sequence
        ,FDV$K_UHELP_ALL=  2002 !Help given, repeat UAR
```

```
!**************************************
! Function Key UAR return codes
!**************************************
    ,FDV$K_UKEY_ERR = 3000 !Fn Key failure, FDV signals
    ,FDV$K_UKEY_TRM = 3001 !Fn Key success, normal F.K.
    ,FDV$K_UKEY_NXT = 3002 !Fn Key succ, treat as NEXT
    ,FDV$K_UKEY_NTR = 3003 !Fn Key succ, treat as ENTER
    ,FDV$K_UKEY_SUC = 3004;!Fn Key succ, ignore
```

Chapter 4
Programming FMS Applications in VAX–11 C

The FMS Form Driver processes all Form Driver calls according to VAX–11 standards and language-specific rules. These rules define how parameters are passed to the Form Driver and how values are returned to your program. Language-specific information is briefly presented in this manual. For more detail, refer to the VAX–11 C document set.

Your VAX–11 C application program must comply with the requirements of the VAX–11 C FMS interface. Topics discussed in this chapter include:

- Invoking Form Driver Routines
- Parameter Passing in FMS
- Null Arguments
- FMS Data Types
  - Character Strings
  - Longword Binary Integers
  - Word Binary Integers
- Descriptors
- Non-FMS Data Types
- One-Dimensional Arrays
- Allocation: Workspace, Terminal Control Area, and Run-Time Memory-Resident Form Area
- Precautions for Using FMS
- Data Conversion
- Sample Application Program in VAX–11 C

A sample program written in C (SAMPCC.C) appears at the end of this chapter. Following the code for SAMPCC.C are Form Driver definition files created for SAMPCC.C. Command file information needed to build the Sample Application program is in Section 4.11.2.

Examples from the Sample Application are used throughout the text to illustrate language issues. Where appropriate examples from SAMPCC.C do not exist, other examples are provided.

4.1 Invoking Form Driver Routines

In the C language, all out-of-line code sequences (such as procedures and subroutines) are called "functions." All C functions are external. The Form Driver routines are called from a C program using the standard C function reference syntax. For example:

```
fdv$wait  ();
```

Calls the Form Driver routine fdv$wait and passes no parameters.

```
fdv$get (&_option, &terminator, $DESCR("OPTION"));
```

Calls the routine fdv$get and passes three parameters.

A status code is returned to the calling program at completion of all Form Driver function calls. To receive the returned status code from a Form Driver function, you activate the routine with a function reference. Note that this returns a standard VMS status code. For portability, other status mechanisms can also be used. (For more information, see the *VAX–11 FMS Form Driver Reference Manual*, Chapter 2.)

See Appendix A for a complete list of Form Driver calls. The calling sequence for each Form Driver routine, data access codes, data types, and passing mechanisms are presented in language-independent notation as specified by the VAX–11 Procedure Calling and Condition Handling Standard. For further detail about the VAX–11 Procedure Calling and Condition Handling Standard, refer to the *VAX–11 Run-Time Library Reference Manual.*

4.2 Parameter Passing in FMS

The parameter passing mechanism refers to the way in which data is passed to a called routine. The VAX–11 Procedure Calling Standard has three methods for passing parameters:

- By reference
- By descriptor
- By value

In C, all arguments are normally passed by value. FMS routines, however, expect arguments to be passed only by reference and by descriptor.

By reference specifies that the storage location of the argument is passed to the routine. FMS expects integer arguments to be passed by reference. From C, you can pass an integer to FMS by specifying its address in the argument list (using the & operator). Alternatively, you can pass a pointer whose current value is the address of the integer.

By descriptor specifies that the address of a descriptor of the argument is passed to the routine. FMS expects character strings and arrays to be passed by descriptor. (A descriptor is a data structure that specifies information about the argument.) The C language has no built-in facility for passing arguments by descriptor. Therefore, descriptors must be explicitly declared, and their fields filled in with the appropriate information, before they can be used to pass arguments to FMS routines. An argument can be passed by specifying the address of its descriptor in the argument list (using the & operator). Alternatively, you can pass a pointer whose current value is the address of the descriptor. See Section 4.5 for more information on descriptors.

4.3 Null Arguments

When the call syntax includes optional parameters and you do not wish to specify all of the information, you can use null arguments. Any optional parameter can be omitted to simplify your program. An address of 0 is assigned to each null argument. Optional parameters to the right of the last required parameter can simply be omitted from the call. In the following example, the fdv$getal call passes only the field terminator value:

```
fdv$getal (0, &terminator);
```

4.4 FMS Data Types

4.4.1 Character Strings

The character string is one of the general data types used by FMS. For example, the fdv$get call passes the character strings for field value (_option) and field name("OPTION"):

```
fdv$get (&_option, &terminator, $DESCR("OPTION"));
```

You must be certain that your strings are initially declared to be long enough to accommodate your FMS data. When passing strings to the Form Driver, be careful to set the length so that the null byte at the end of the string is not overwritten. It is necessary to provide a descriptor for every string that you wish to pass to FMS. See Section 4.5.2 for information on constructing and using string descriptors.

It is possible to use a single string variable in different FMS calls to transfer data to or from several forms and fields. You must declare the string variable to be at least as large as the longest field value string that will be returned to the program. Use the fdv$retle call to return the length of the valid portion of the field value in the string variable. This length can then be used when referencing the data that was entered in the field.

```
fdv$get (&_account, &terminator, $DESCR("FIELD"));
fdv$retle (&lengthfield, $DESCR("FIELD"));
```

After the execution of the fdv$retle call, lengthfield is equal to the length of the field named "FIELD". It is also equal to the valid portion of the string that is defined by the string descriptor _account. The variable lengthfield is used when referencing the data that was entered in the field named "FIELD", and which is now in the variable _account.

A useful application of the fdv$retle call is in general purpose user action routines.

4.4.2 Longword Binary Integers

The longword binary integer is another general data type used by FMS. For example, the fdv$aterm call passes the longword value for terminal control area size (12) and logical I/O channel number (2):

```
fdv$aterm (&_tcarea, &12, &2);
```

Numeric parameters must be longword binary integers. If you try to pass other numeric types to the Form Driver, the calls do not work properly. An exception is the fdv$dfkbd call (see the following section).

4.4.3 Word Binary Integers

The defkbd argument is a word integer array passed when the FDV$DFKBD routine is invoked. FMS expects arrays to be passed by descriptor.

4.5 Descriptors

A descriptor is a VMS data structure that provides a mechanism for communicating data between independent procedures in an indirect, uniform, and controlled fashion. It contains all the information needed to characterize any given data item. FMS, as a VMS-layered product, uses descriptors, but the C language has no built-in facility for passing arguments by descriptor. To pass arguments by descriptor from a C program to an FMS routine, you must construct and manipulate your own descriptors.

The format and content of various descriptors is detailed in the *VAX–11 Run-Time Library Reference Manual*. As an aid to using descriptors in C,

VAX–11 C provides an include file, DESCRIP.H, which contains the same information in the form of structure declarations and macro definitions. To access this information, simply include the following line in your program:

```
#include <descrip.h>
```

4.5.1 Passing Arguments by Descriptor

You pass an argument by descriptor as follows:

1. Write a structure declaration that models the desired descriptor.
2. Set the appropriate values in the descriptor fields. This can be done either by specifying initial values for the structure members right in the declaration, or by setting the values at run time with explicit assignment statements.
3. Using the ampersand operator (&), specify the address of the structure in the argument list to the FMS function.

For example:

```
/* Include the canned descriptor declarations */
#include <descrip.h>
    .
    .
    .
/* Declare and initialize some data to be passed to FMS */
char form_name [6] = "FORM1";
    .
    .
    .
/* Declare and initialize a descriptor for the above data */
$DESCRIPTOR (dsc_form, form_name);
    .
    .
    .
/* Pass the data to FMS */
fdv$cdisp (&dsc_form);
```

4.5.2 String Descriptors

The most common descriptor you will be using is an ordinary string descriptor, which can be declared as a simple structure with four fields. For example, DESCRIP.H defines a string descriptor as follows:

```
struct dsc$descriptor_s
      {
      unsigned short    dsc$w_length;
      unsigned char     dsc$b_dtype;
      unsigned char     dsc$b_class;
      char              *dsc$a_pointer;
      }
```

These four fields, common to all descriptors, have the following meanings for string descriptors:

- dsc$w_length The length of the string to be passed
- dsc$b_dtype A code indicating what kind of string is to be passed
- dsc$b_class A code indicating that this is a string descriptor
- *dsc$a_pointer A pointer to the character string

To simplify declaring and initializing string descriptors, DESCRIP.H defines a preprocessor macro, $DESCRIPTOR. $DESCRIPTOR constructs a structure declaration with two arguments. The first argument is an identifier specifying the name of the descriptor to be declared and initialized. The second argument is used to initialize the length and pointer fields of the descriptor. This argument can be either a literal string constant or the name of a variable previously declared as an array of characters. The macro expansion itself initializes the data type and class fields for you.

4.5.3 Macros

C users typically use macros to provide a level of abstraction that does not exist in the language. For example, it is easier to use $DESCRIPTOR than to code out the entire sequence. To view what the compiler actually substitutes into the text of the program after the expansion of the macro, you can specify the CC command line qualifier /SHOW = EXPANSION. This will print the results of the macro expansion in your listing. The expansion of the macro is useful as a debugging aid. For example, the $DESCRIPTOR macro used in the example in Section 4.5.2 above expands to the following declaration:

```
struct dsc$descriptor_s dsc_form = {sizeof(form_name)-1, 14, 1, form_name};
```

(Note that in the Sample Application, there are some situations where the $DESCRIPTOR macro is not adequate. Thus, other similar macros are defined to be used by the program to declare its descriptors. These other macros are not supplied with DESCRIP.H.) As an alternative to $DESCRIPTOR, you can declare the descriptor without initialization, providing instead for setting the descriptor field values at run time. This can be useful if you want to reuse a descriptor for many different pieces of data. For example:

```
#include <descrip.h>
  .
  .
  .
/* Declare a descriptor, using the structure tag from DESCRIP.H */
struct dsc$descriptor_s dsc_form;
  .
  .
  .
```

```
/* Set the data type and descriptor class fields */
dsc_form.dsc$b_dtype = DSC$K_DTYPE_T;
dsc_form.dsc$b_class = DSC$K_CLASS_S;
   .
   .
   .
/* Set the string length and address fields */
dsc_form.dsc$w_length = 5;
dsc_form.dsc$a_pointer = FORM1'';
   .
   .
   .
/* Pass the address of the descriptor to FMS */
fdv$cdisp (&dsc_form);
```

The names DSC$K_DTYPE_T and DSC$K_CLASS_S are macros defined in DESCRIP.H. See the *VAX–11 Run-Time Library Reference Manual* or a listing of DESCRIP.H for more information.

4.6 Non-FMS Data Types

C data types that are not recognized by FMS can be used in your C application program provided they are not passed to the Form Driver.

4.7 One-Dimensional Arrays

One-dimensional arrays are structures that can be used in FMS for the following arguments:

- tca (terminal control area)
- wksp (workspace)
- mloc (memory location)
- defkbd (define keyboard)

You must provide FMS with storage space for these arguments. You can do that by defining them to be:

- longword integer arrays or character strings for tca, wksp, and mloc
- word integer arrays for defkbd

In the Sample Application program, the tca, wksp, and mloc arguments are passed to several Form Driver routines. In the declaration section of the program, these arguments are defined to be integer array variables. You may alternatively define these variables to be character strings in your own application program. Use of character strings rather than longword integer arrays avoids the need to work with array descriptors which are considerably more complicated than string descriptors.

You can use macros to set up descriptors for the integer arrays passed by your program. The Sample Application program uses the following macro, created for the sample program, to declare longword integer arrays.

```
/*
* $DESCRIPTORA generates an array descriptor, and is used to describe
*              the workspaces and terminal control area
*/

#define $DESCRIPTORA(name,array,type)  struct dsc$descriptor_a \
name = { sizeof(type), DSC$K_DTYPE_L, DSC$K_CLASS_A, \
array, 0, 0, { 0, 0, 0, 0, 0 }, 1, sizeof array }
```

The following Sample Application program declarations establish names and storage for the integer array variables workspace, checkwksp, tcarea, and menu_form:

```
static  int
        workspace [3],     /* General workspace              */
        checkwksp [3],     /* Check workspace                */
        tcarea [3],        /* Terminal Control Area          */
        menu_form [500];   /* Storage for memory-resident form */

/* Array descriptors for above */
static  $DESCRIPTORA (_workspace, workspace, int);
static  $DESCRIPTORA (_checkwksp, checkwksp, int);
static  $DESCRIPTORA (_tcarea, tcarea, int);
static  $DESCRIPTORA (_menu_form, menu_form, int);
```

4.8 Allocation: Workspace, Terminal Control Area, and Run-Time Memory-Resident Form Area

The FMS workspace, terminal control area, and run-time memory-resident form area variables are character strings or longword integer arrays. These variables should be placed in a static or external storage area of your program. Note that this is not done in the Sample Application program. The sample program's structure protects the workspaces, terminal control areas, and run-time memory-resident form areas implicitly.

The Form Driver needs 12 bytes to associate user information about each workspace and terminal control area. The allocation for a run-time memory-resident form area must be the size of the form. The space for all of these variables is allocated by your application program. Note that FMS uses *only* 12 bytes of space for the workspace and terminal control area allocation. Any allocation greater than 12 bytes is wasted. You declare your workspace, terminal control area, and run-time memory-resident form area only once. FMS remembers their addresses after the addresses have been initially passed to the Form Driver.

For each workspace, the Form Driver also allocates an additional amount of storage space based on your estimate of the amount of memory needed to store your largest form. If your estimate is too small, the Form Driver

allocates more space automatically, but performance may be affected. An adequate estimate results in more efficient operation of the Form Driver. You can use the FMS/DIRECTORY/FULL command to find out how much space to allocate.

In the following example from the Sample Application program, the workspace is allocated and the fdv$awksp routine is called. When the fdv$awksp routine is called, the first argument (_workspace) specifies the area of memory to be used for your workspace. The second argument specifies an estimate of the workspace size (2000 bytes) that you will need to display the largest form in your application.

```
fdv$awksp (&_workspace, &2000);
```

4.9 Precautions for Using FMS

4.9.1 Memory Areas Used Exclusively by FMS

The locations for terminal control area, workspace, and run-time memory-resident form area are used exclusively by FMS. The terminal control area and workspace are attached with the FDV$ATERM and FDV$AWKSP calls and remain allocated until the FDV$DTERM and FDV$DWKSP calls are issued or until the program ends. The run-time memory-resident form area, used in the FDV$READ call, remains allocated until the FDV$DEL call is issued or until the program ends. You never touch the terminal control areas, workspaces, or run-time memory-resident form areas in your program except to pass their addresses to the Form Driver.

4.9.2 Why You Should Use Static or External Storage Areas

Parameters to the following Form Driver routines should be used with caution:

fdv$aterm	Attach terminal
fdv$awksp	Attach form workspace
fdv$read	Read form into memory
fdv$ssrv	Specify status reporting variables

For example, once an fdv$ssrv call is issued, the program variables that contain the FMS status and RMS status become volatile and can change at any call point. As a general rule, you should place the status variables in static or external storage.

In cases where you need both the FMS and RMS statuses, the fdv$stat routine can be used. Note that only the fdv$stat and fdv$ssrv calls provide RMS status. With the fdv$stat routine, you do not have to worry about volatility.

The locations for terminal control area, workspace, run-time memory-resident form area, and status reporting variables must all continue to exist while the Form Driver is using them. They must remain allocated until the terminal control area and workspace are detached, until forms in memory location are deleted, and until the status reporting variables are no longer used. The variables can be protected by placing them in external storage; otherwise, the compiler might place them in dynamic storage.

4.10 Data Conversion

FMS uses only ASCII character strings to display data. All information displayed on the terminal screen, and all information received from the terminal operator, is represented as ASCII field values. Any manipulation of numeric data requires conversion of ASCII character strings to numeric data, and conversion of numeric data back to ASCII character strings. In the following discussion of conversion routines, you should assume that the receiving data type can support the largest number that is likely to be generated.

User-created functions are useful to perform the conversions necessary to accommodate FMS requirements. The Sample Application creates several data conversion functions.

In the Sample Application, the following steps are taken to get a new account balance after writing a check:

```
/*
 * Get amount from check,
 * Update balance (in memory and on screen) and session sums,
 */
fdv$ret ($DESCR2 (regarray [lastregnum].ri_amtpay), $DESCR ("AMTPAY"));
amtpay = val (regarray [lastregnum].ri_amtpay,
              sizeof regarray[0].ri_amtpay);
balance -= amtpay;
totpay += amtpay;

fdv$put (itoa (balance), $DESCR ("BALANCE"));
```

In this example, the user-created function val reads the character string named regarray [lastregnum].ri_amtpay and returns the numeric value as a longword integer. The integer value is assigned to the variable amtpay. The integer value of the variable amtpay is subtracted from the integer value of the variable balance to produce a new balance. The value of amtpay is added to the integer value of the variable totpay to produce a new value for totpay.

After the data operations have been completed, the user-created function itoa converts the integer value of the variable balance to the corresponding ASCII character string descriptor. The value for balance is displayed in the right-justified field "BALANCE". The rightmost digit from the program is

displayed in the field's rightmost character position. The remaining digits of the character expression are placed to the left of the rightmost digit. If output is longer than the field, FMS truncates on the left. (The Form Driver displays a data length error message (FDV$_DLN) only if you have set FMS Debug mode.)

For other conversion options, see the general conversion routines in the *VAX–11 Run-Time Library Reference Manual.*

4.11 Sample Application Program in VAX–11 C

The FMS Sample Application program (SAMPCC.C) is part of the FMS distribution kit. When FMS is installed, SAMPCC.C is placed in the directory FMS$EXAMPLES. Designed to be a demonstration program and learning tool, SAMPCC.C shows most of the features provided by FMS. The entire Sample Application program appears at the end of this chapter.

4.11.1 Form Driver Definition Files

The include file FDVDEF.H is part of the Sample Application program package. When FMS is installed, FDVDEF.H is placed in the directory FMS$EXAMPLES. The FDVDEF.H file appears after the Sample Application source code.

FDVDEF.H contains a variety of codes for the Form Driver routines used in the Sample Application program. Although these codes have been created for use in SAMPCC.C, they can provide you with a helpful starting point as you create definitions for your own application program. The file FDVDEF.H includes:

- FMS terminator codes
- Function key terminators returned from the FDV$GET and FDV$WAIT calls
- Form Driver key functions for use with the FDV$DFKBD call
- User action routine (UAR) return codes, which are returned by the UARs to the Form Driver:

  Field completion UAR return codes

  Help UAR return codes

  Function key UAR return codes
- VMS status codes returned when Form Driver routines are called as functions. These codes can be signaled.
- FMS status codes returned when the FDV$STAT routine is called as a function
- Declarations of Form Driver routines

4.11.2 Command File for Building the Sample Application Program

The command file for building the Sample Application program includes all the information that you need to compile and link SAMPCC.C. When FMS is installed, the command file is placed in the directory FMS$EXAMPLES.

```
$!       S A M P C C . C O M
$!
$!       Compile and link the C version of the FMS V2 Sample Application
$!
$!       The C source files are:          SAMPCC.C
$!                                        FDVDEF.H
$!
$!       SMPVECTOR.OBJ and SMPMEMRES.OBJ were produced by the FMS commands:
$!
$!         $ FMS/VECTOR/OUTPUT=SMPVECTOR SAMP.FLB
$!         $ FMS/MEMORY/OUTPUT=SMPMEMRES SAMP.FLB/FORM=(HELP_KEYS,HELP_MENU)
$!
$ CC     SAMPCC
$ LINK   SAMPCC, FMS$EXAMPLES:SMPVECTOR, FMS$EXAMPLES:SMPMEMRES, -
                 SYS$LIBRARY:CRTLIB/LIBRARY
```

```
/*
 ************************************************************************
 *                                                                      *
 * SAMP -- The FMS Sample Application Program in VAX-11 C               *
 *                                                                      *
 ************************************************************************
 */

#include stdio                    /* Definitions for I/O              */
/*
 * C has no built-in facility for passing arguments by descriptor, so we
 * have to define and set up all descriptors explicitly.  The following file
 * contains structure definitions for the various descriptor types:
 */
#include descrip

/*
 * A macro, $DESCRIPTOR, is defined in the file included above, and can be
 * used to declare descriptors for the usual C strings (NUL-terminated arrays
 * of char).  This macro is not adequate for all the cases requiring descriptors
 * in this program, however, so we must define a few more macros:
 *
 * $DESCRIPTOR1 can be used for 1-character items, or for whole structures
 * $DESCRIPTORM can be used for strings that are structure members, or for
 *              any other string not terminated by a NUL character
 * $DESCRIPTORA generates an array descriptor, and is used to describe
 *              the workspaces and terminal control area
 *
 * LENGTH and POINTER are useful shorthands for accessing descriptor fields
 */

#define $DESCRIPTOR1(name,string)       struct dsc$descriptor_s
name = { sizeof(string), DSC$K_DTYPE_T, DSC$K_CLASS_S, string }

 define $DESCRIPTORM(name,string)       struct dsc$descriptor_s
name = { sizeof(string), DSC$K_DTYPE_T, DSC$K_CLASS_S, string }

#define $DESCRIPTORA(name,array,type)   struct dsc$descriptor_a
name = { sizeof (type), DSC$K_DTYPE_L, DSC$K_CLASS_A, array, 0, 0, { 0, 0, 0, 0, 0 }, 1, sizeof array }

#define LENGTH(descriptor)      descriptor.dsc$w_length
#define POINTER(descriptor)     descriptor.dsc$a_pointer
```

```
/*
 * Data definitions
 */


/*
 * FMS related
 */

static  int
        workspace [3],          /* General workspace            */
        checkwksp [3],          /* Check workspace              */
        tcarea [3],             /* Terminal Control Area        */
        menu_form [500],        /* Storage for memory resident form */
        size_menu,
        check_form [750],       /* Storage for memory resident form */
        size_check,
        dposit_form [500],      /* Storage for memory resident form */
        size_dposit;

/* Array descriptors for above  */
static  $DESCRIPTORA (_workspace, workspace, int);
static  $DESCRIPTORA (_checkwksp, checkwksp, int);
static  $DESCRIPTORA (_tcarea, tcarea, int);
static  $DESCRIPTORA (_menu_form, menu_form, int);
static  $DESCRIPTORA (_check_form, check_form, int);
static  $DESCRIPTORA (_dposit_form, dposit_form, int);


/*
 * Account (Read in from file)
 */

static  struct  {
                char    acctno [5];
                char    accdate [7];
                char    last [20];
                char    first [15];
                char    middle [15];
                char    street [30];
                char    city [20];
                char    state [2];
```

```
                char    zip [5];
                char    homeph [10];
                char    workph [10];
                char    opw [12];
                char    account_nul [2];        /* Space for NL and NUL (C only) */
                } account;


/*
 * Deposit data (Read via fdv$setal)
 */

static  struct  {
                char    dep_date [7];
                char    dep_curbal [6];
                char    dep_amt [6];
                char    dep_newbal [6];
                char    dep_memo [35];
                } deposit;
/*
 * Money.
 * Note that all money is kept internally as integers (in cents).
 * It is only when the quantities are output that they look like
 * dollars, since all the money fields have periods as field
 * markers in the right places and they are right justified or
 * fixed decimal.
 *
 * Register data.
 */

#define REGSIZE 30

static  struct  {
                char    ri_num [4];
                char    ri_date [7];
                char    ri_mempayto [35];
                char    ri_amtdep [6];
                char    ri_amtpay [6];
                char    ri_balance [6];
                char    ri_nul [2];                 /* Space for NL and NUL (C only) */
                } regarray [REGSIZE + 1];
```

```
/*
 * Other variables
 */

static  int
        terminator,     /* Terminator returned by FDV */
        balance,        /* Balance in account, numeric */
        sbalance,       /* Starting balance */
        totdep,         /* Total deposits made in this session */
        totpay,         /* Total checks payed in this session */
        fmsstatus,      /* Status for last FDV call */
        rmsstatus,      /* RMS Status for last FDV call */
        lastresnum,     /* Last number used in the register (1...REGSIZE) */
        lastchnum,      /* Last check number used */
        curline,        /* Line of check register that cursor is now on */
        minwindow,      /* Smallest line of register being displayed
                           on the scrolled area */
        maxwindow;      /* Largest line of register being displayed
                           on the scrolled area */
/*
 * We could use a couple of string descriptors for temporary use
 * and macros to make it easier to pass literal strings to FMS
 */
static  struct  dsc$descriptor_s
        _temp_descr_1 = { 0, DSC$K_DTYPE_T, DSC$K_CLASS_S, 0 },
        _temp_descr_2 = { 0, DSC$K_DTYPE_T, DSC$K_CLASS_S, 0 };

#define $DESCR(literal) (LENGTH (_temp_descr_1) = sizeof literal - 1,\
 POINTER (_temp_descr_1) = literal, &_temp_descr_1)

#define $DESCR2(string) (LENGTH (_temp_descr_2) = sizeof (string),\
 POINTER (_temp_descr_2) = string, &_temp_descr_2)
/*
 * Some string function declarations.
 */
struct  dsc$descriptor_s        *itoa ();

char    *strchr ();
/*
 * FMS terminator codes:
 */
#include "fdvdef.h"
```

```
main ()
{
        /*
         * Initialize FMS
         *   Attach default terminal
         *   Attach normal and check workspaces (order important for help
         *    and refresh during CHECK/CHECK_DONE time--try switching and see).
         *   Open form library, attach to channel 1
         *   Set keypad mode to application
         *   Set signal mode to bell (default, but it's fun to do)
         */
        fdv$aterm (&_tcarea, &12, &2);
        getsta ();
        fdv$awksp (&_checkwksp, &2000);
        getsta ();
        fdv$awksp (&_workspace, &2000);
        getsta ();
        fdv$lopen ($DESCR ("FMS$EXAMPLES:SAMP"), &1);
        getsta ();
        fdv$spaca (&1);
        fdv$ssisg (&0);

        /*
         * Set all future calls to return status to the two status recording
         * variables fmsstatus and rmsstatus without having to call the
         * fdv$stat routine.
         */
        fdv$ssrv (&fmsstatus, &rmsstatus);

        /*
         * Read in a few forms from the form library onto the dynamic
         * resident form list.  You may be able to detect the difference
         * in the form to form access times for those forms which have to be
         * accessed from the form library on disk and those forms which are
         * on the dynamic or static memory resident form list.  See the
         * installation notes for this program (the LINK command) to see
         * which forms are on the static memory resident form list.
         */
        fdv$read ($DESCR ("MENU"), &_menu_form, &2000, &size_menu);
        fdv$read ($DESCR ("CHECK"), &_check_form, &3000, &size_check);
        fdv$read ($DESCR ("DEPOSIT"), &_deposit_form, &2000, &size_deposit);
```

```
        /*
         * Initialize account information
         */
        inacct ();

        /*
         * Put up welcome form, wait for response
         */
        fdv$cdisp (%DESCR ("WELCOME"));
        sync$ ();
        fdv$wait ();

        /*
         * Process all menu requests
         */
        menu ();

        /*
         * Clean up and leave:
         *   Close form library.
         *   Reset keypad to numeric.
         *   Delete a form from dynamic mem. res. form list just to show how.
         *   Detach workspaces (not really necessary since DTERM would do it).
         *   Detach terminal.
         */
        fdv$lclos ();
        fdv$spada (&0);
        fdv$del (%DESCR ("MENU"));
        fdv$dwksp (&_workspace);
        fdv$dwksp (&_checkwksp);
        fdv$dterm (&_tcarea);

}
```

```
/*
 * Subroutine INACCT
 *      Read from file SAMP.DAT into internal variables.
 *      Set up the workspace for checks and fill in the check form
 *      with the account's name, address, and account number.
 */

inacct ()
{
        FILE    *account_file;


        /*
         * Open file, get account data
         */
        if ((account_file = fopen ("FMS$EXAMPLES:samp.dat", "r")) == NULL)
        {
                printf ("Unable to open account file, \"samp.dat\"");
                exit (1);
        }
        fsets (&account, sizeof account, account_file);

        /*
         * Read the remaining records into the check register, counting them.
         * The last record has the current balance, and some record has the
         * last check number used (not necessarily the last record).
         */
        lastchnum = 0;
        lastresnum = 0;
        while (lastresnum < REGSIZE)
        {
                if (fsets (& resarray [lastresnum + 1], sizeof resarray [0], account_file) == NULL)
                        break;
                ++lastresnum;
                if (strncmp (resarray [lastresnum].ri_num, "    ", 4) != 0)
                        lastchnum = val (resarray [lastresnum].ri_num,
                                         sizeof resarray[0].ri_num);
        }

        /*
         * Reached here with or without hitting end of file.
         * If not end of file, should probably print a message or something,
```

```
         * except that this is just a l'il ol' demo.
         * As it is, just fall through and ignore remaining records.
         *
         * Check for data file in error.
         * Take balance from last record read.
         * Set session sums to zero to say no activity yet.
         */
        fclose (account_file);
        if (lastresnum == 0)
        {
                printf ("DATA FILE IN ERROR");
                exit (1);
        }
        balance = val (resarray [lastresnum].ri_balance,
                       sizeof resarray[0].ri_balance);
        sbalance = balance;
        totdep = 0;
        totpay = 0;
        /*
         * Set up the check workspace once so we don't have to do it every time.
         */
        fmtchk ();

        return;

}



/*
 * Subroutine FMTCHK
 *      Format account data onto check form in the check workspace.
 */

fmtchk ()
{
        static  char    strins [sizeof account.first + sizeof account.middle +
                                sizeof account.last + 3];
        static  $DESCRIPTOR (_strins, strins);
        $DESCRIPTORM (_first, account.first);
        $DESCRIPTORM (_middle, account.middle);
```

```
$DESCRIPTORM (_last, account.last);
$DESCRIPTORM (_street, account.street);
$DESCRIPTORM (_city, account.city);
$DESCRIPTORM (_homeph, account.homeph);
$DESCRIPTORM (_acctno, account.acctno);


fdv$swksp (&_checkwksp);
fdv$load ($DESCR ("CHECK"));

trim (&_first);
LENGTH (_middle) = 1;
trim (&_last);
sprintf (string, "%.*s %.*s. %.*s",
         LENGTH (_first), POINTER (_first),
         LENGTH (_middle), POINTER (_middle),
         LENGTH (_last), POINTER (_last) );
LENGTH (_string) = strlen (string);
fdv$put (&_string, $DESCR ("NAME"));

fdv$put (&_street, $DESCR ("STREET"));

trim (&_city);
sprintf (string, "%.*s, %.*s %.*s",
         LENGTH (_city), POINTER (_city),
         sizeof account.state, account.state,
         sizeof account.zip, account.zip );
LENGTH (_string) = strlen (string);
fdv$put (&_string, $DESCR ("CSZ"));

fdv$put (&_homeph, $DESCR ("HOMEPH"));

fdv$put (&_acctno, $DESCR ("ACCTNO"));

fdv$swksp (&_workspace);

}
```

```
/*
 * Subroutine MENU
 *      Accept inputs from the menu form and dispatch to the
 *      appropriate routine.  Repeat until option 1 (exit) is
 *      chosen.  The UARs in the form guarantee that we get back
 *      only inputs '1'-'5' with the correct terminators.
 *      Options are:
 *              1 => Exit
 *              2 => Write checks
 *              3 => Make deposit
 *              4 => View register
 *              5 => View account data
 */

menu ()
{
        static  char    option;
        static  $DESCRIPTOR1 (_option, option);


        while (1)
        {
                fdv$coisp ($DESCR ("MENU"));
                srvchk ();
                fdv$set (&_option, &terminator, $DESCR ("OPTION"));

                switch (option)
                {
                case '1':
                        return;

                case '2':
                        writch ();
                        continue;

                case '3':
                        makdep ();
                        continue;

                case '4':
                        vueres ();
                        continue;
```

```
                case '5':
                        vueact ();
                        continue;
                }

        }

}



/*
 * Subroutine WRITCH
 *      Write one or more checks
 */

writch ()
{
        /*
         * Turn on LED 3 on the VT100 during this routine, just to show how.
         */
        fdv$ledon (&3);

        /*
         * Mark WORKSPACE not displayed so it doesn't show up during refresh.
         * Put up CHECK form from already loaded workspace
         * and display current balance
         */
        fdv$ndisp ();
        fdv$swksp (&_checkwksp);
        fdv$dispw ();

        fdv$put (itoa (balance), $DESCR ("BALANCE"));

        /*
         * Process checks until a keypad period is read
         */
        do {
                onechk ();      /* Process one check */
                endchk ();      /* Give options for continuing */
        } while (terminator != FDV$K_KP_PER);
```

```
        /*
         * Turn off LED 3 on VT100
         */
        fdv$ledof (&3);
        fdv$swksp (&_workspace);

}



/*
 * Subroutine ONECHK -- Process one check
 *      If input is terminated by kpd period, return with no action
 *      Else deduct from balance and enter into register.
 *      Note that a UAR in the form guarantees that the amount of
 *      the check is always less than or equal to the balance.
 *      Note that the form function key UAR  allows only kpd period
 *      as terminator (other than FDV$K_FT_NTR).
 */

onechk ()
{
        static  char    junk [2];
        static  $DESCRIPTOR (_junk, junk);
        int     amtpay;

        fdv$put (itoa (lastchnum + 1), $DESCR ("NUMBER"));
        fdv$getal (&_junk, &terminator);
        if (terminator == FDV$K_KP_PER)
                return;

        /*
         * If the check wouldn't fit in the register, don't process, just
         * give error message, wait for acknowledgement, and return
         */
        if (lastregnum == REGSIZE)
        {
                fdv$putl ($DESCR ("Register full, can't enter check"));
                fdv$wait ();
                return;
        }
```

```
        ++lastresnum;

        /*
         * Get amount from check.
         * Update balance (in memory and on screen) and session sums.
         */
        fdv$ret ($DESCR2 (resarray [lastresnum].ri_amtpay), $DESCR ("AMTPAY"));
        amtpay = val (resarray [lastresnum].ri_amtpay,
                      sizeof resarray[0].ri_amtpay);
        balance -= amtpay;
        totpay += amtpay;

        fdv$put (itoa (balance), $DESCR ("BALANCE"));
        fdv$ret ($DESCR2 (resarray [lastresnum].ri_balance), $DESCR ("BALANCE"));

        strncpy (resarray [lastresnum].ri_amtdep, "      ", 6);
        fdv$ret ($DESCR2 (resarray [lastresnum].ri_num), $DESCR ("NUMBER"));
        fdv$ret ($DESCR2 (resarray [lastresnum].ri_date), $DESCR ("DATE"));
        fdv$ret ($DESCR2 (resarray [lastresnum].ri_mempayto), $DESCR ("PAYTO"));
                                            /* Note: not from check's MEMO */

        ++lastchnum;

}



/*
 * Subroutine ENDCHK
 *      Finish off check processing by giving operator
 *      three options:
 *              RETURN  Write another check
 *              KPD 0   Print the check into file SAMPCH.DAT
 *              KPD .   Return to menu
 *      Check to see if check write was aborted by kpd per.
 *      If so, then don't give any further choice, just abort.
 *      Note that form function key UAR allows only the above
 *      terminators to get through.
 */
```

```
endchk ()
{
        if (terminator == FDV$K_KP_PER)
                return;

        /*
         * Tell the operator that the check has been paid by overlaying with
         * a new form, using the normal workspace, thereby saving the check
         * workspace in case another check is to be written.
         */
        fdv$swksp (&_workspace);
        fdv$disp ($DESCR ("CHECK_DONE"));
        srvchk ();

        /*
         * Wait for operator to enter either KPD period, NTR, or KPD zero.
         * Print the check as many times as requested.
         * (Note that a UAR on the form guarantees that only those terminators
         * are accepted).
         * Process accordingly.
         */
        fdv$wait (&terminator);
        while (terminator == FDV$K_KP_0)
        {
                prichk ();      /* Print the check */
                fdv$wait (&terminator);
        }

        /*
         * If choice is to quit,
         * then mark check wksp undisplayed so it doesn't appear during refresh,
         * else mark normal workspace (occupied by CHECK_DONE form) undisplayed
         *   so it doesn't show during refesh and then clear its lines.
         *   (Clearing the space occupied by the CHECK_DONE form, lines 20-23
         *   is better done by overlaying with a blank form to
         *   avoid having to know the line numbers to clear).
         */
        if (terminator == FDV$K_KP_PER)
        {
                fdv$swksp (&_checkwksp);
                fdv$ndisp ();
        }
```

```
        else
        {
                fdv$ndisp ();
                fdv$clear (&20, &4);
                fdv$swksp (&_checkwksp);
        }

        /*
         * Going to write another check now or eventually, so:
         * Clear out operator entered fields.
         */
        fdv$putd ($DESCR ("AMTPAY"));
        fdv$putd ($DESCR ("MEMO"));
        fdv$putd ($DESCR ("PAYTO"));

}



/*
 * Subroutine PRICHK
 *      Print the check into the file SAMPCH.DAT
 *      Use the check workspace, then switch back to the normal wksp
 *      to keep things clean.
 */

prichk ()
{
        static  char    firstl [3],     /* First line on the form of the check
                                           image (from named data) */
                        lastl [3],      /* Last line on the form of the check
                                           image (from named data) */
                        line [81];      /* Line return as image of form for
                                           check print */
        static  $DESCRIPTOR (_firstl, firstl);
        static  $DESCRIPTOR (_lastl, lastl);
```

```
        static  $DESCRIPTOR (_line, line);
        FILE    *check_file;
        int     i,      /* Index into lines of check */
                line_length;    /* length of form line */


        /*
         * Open check writing file.  Note there's a new version for every check.
         * Switch workspaces
         */
        check_file = fopen ("sampch.dat", "w");
        fdv$swksp (&_checkwksp);

        /*
         * Get the top and bottom lines of the check from the named data
         * (first two characters).
         */
        fdv$retdn ($DESCR ("FIRST"), &_firstl);
        srvchk ();
        fdv$retdn ($DESCR ("LAST"), &_lastl);
        srvchk ();

        /*
         * Get lines from form.
         * Convert to line printer style.
         * Write to file.
         */
        for (i = atoi (firstl);  i <= atoi (lastl);  ++i)
        {
                fdv$retfl (&i, &_line, &line_length, &0);
                fprintf (check_file, "%s\n", line);
        }
        fdv$putl ($DESCR ("Check written to file"));
        fclose (check_file);
        fdv$swksp (&_workspace);

}
```

```
/*
 * Subroutine MAKDEP
 *      Make a deposit, enter into check register
 *      Cancel on keypad period.
 *      Note that the form function key UAR allows only kpd period.
 *
 * Put up deposit form with current balance
 */

makdep ()
{
        static  char    done [80];      /* Form done message for Deposit */
        static  $DESCRIPTOR (_done, done);
        static  $DESCRIPTOR (_deposit, deposit);


        fdv$cdisp ($DESCR ("DEPOSIT"));
        srvchk ();
        fdv$put (itoa (balance), $DESCR ("CURBAL"));

        /*
         * Get deposit amount and memo from operator.
         * Abort on kpd period.
         */
        fdv$setal (&_deposit, &terminator);
        if (terminator == FDV$K_KP_PER)
                return;

        /*
         * Have deposit information now. If no room in check register
         * must abort.
         */
        if (lastresnum == REGSIZE)
        {
                fdv$putl ($DESCR ("Register full, can't enter deposit"));
                fdv$wait ();
                return;
        }
        ++lastresnum;
```

```
        /*
         * Add to balance and session sum.
         * Check for overflow (program and form keep only six digits).
         * Display new balance.
         * Make entry in register.
         */
        balance += val (deposit.dep_amt, sizeof deposit.dep_amt);
        totdep += val (deposit.dep_amt, sizeof deposit.dep_amt);
        if (balance >= 1000000)
        {
                balance -= 1000000;
                fdv$putl ($DESCR ("Overflow in bank computer, \
only 6 digits allowed, we keep the rest of the money"));
                fdv$wait ();
        }
        fdv$put (itoa (balance), $DESCR ("NEWBAL"));
        /* Blank since it's not a check: */
        strncpy (resarray [lastresnum].ri_num, "    ", 4);
        strncpy (resarray [lastresnum].ri_date, deposit.dep_date, sizeof deposit.dep_date);
        strncpy (resarray [lastresnum].ri_mempayto, deposit.dep_memo, sizeof deposit.dep_memo);
        strncpy (resarray [lastresnum].ri_amtdep, deposit.dep_amt, sizeof deposit.dep_amt);
        strncpy (resarray [lastresnum].ri_amtpay, "      ", 6);
        fdv$ret ($DESCR2 (resarray [lastresnum].ri_balance), $DESCR ("NEWBAL"));

        /*
         * Sample of how to keep message texts stored with the form rather
         * than in a program.  This is especially useful for multi-lingual
         * environments: only the form text and the form named data must
         * be changed and nothing in the program.  The trick is to store the
         * response text in named data. This is the only example of how to do
         * it in this program, but all messages could be stored like this.
         * Message intent is: "Deposit made, press RETURN or ENTER to continue."
         */
        fdv$retdn ($DESCR ("DONE"), &_done);
        fdv$putl (&_done);
        fdv$wait ();

}
```

```
/*
 * Subroutine VUEREG
 *      View the check register and scroll through it.
 *      Also display totals for current session.
 *
 * Put up register form.
 * Check for current session totals overflow. If so, output "OVRFLO"
 * Put out summary of this session into indexed(4) fields.
 */

vuereg ()
{
        static  char    nscrols [3],    /* Number of lines in scrolled area
                                           (from named data) */
                        fake;           /* Value returned from fake field
                                           in scrolled area */
        static  $DESCRIPTOR (_nscrols, nscrols);
        static  $DESCRIPTOR1 (_fake, fake);
        static  $DESCRIPTOR (_sumary, "SUMMARY");
        static  struct  dsc$descriptor_s
                _reg_temp_desc = { 0, DSC$K_DTYPE_T, DSC$K_CLASS_S, 0 };
        int     nscrol;


        fdv$cdisp ($DESCR ("REGISTER"));
        srvchk ();
        fdv$put (itoa (sbalance),       &_sumary, &1);
        fdv$put (totdep < 1000000 ? itoa (totdep) : $DESCR ("OVRFLO"),
                                        &_sumary, &2);
        fdv$put (totpay < 1000000 ? itoa (totpay) : $DESCR ("OVRFLO"),
                                        &_sumary, &3);
        fdv$put (itoa (balance),        &_sumary, &4);

        /*
         * Get number of lines in scroll area from form named data (item 1).
         */
        fdv$retdi (&1, &_nscrols);
        srvchk ();
        nscrol = atoi (nscrols);
```

```
        /*
         * Put lines from check register array into scrolled area.
         * The window is initially from item 1 up to item
         * min(nscrols,lastresnum), that is, up to the size of the scrolled
         * area or the size of the register, whichever is less. Assume there
         * is at least one line (the initial deposit).
         */
        minwindow = 1;
        LENGTH (_res_temp_descr) = sizeof resarray [0] - 2;
        POINTER (_res_temp_descr) = & resarray [1];
        fdv$putsc ($DESCR ("NUMBER"), &_res_temp_descr);        /* First line */
        curline = 1;                                  /* Res item cursor is on */
        while (curline < lastresnum  &&  curline < nscrol)
        {
                ++curline;
                fdv$pft (&FDV$K_FT_SFW, $DESCR ("NUMBER"));
                POINTER (_res_temp_descr) = & resarray [curline];
                fdv$putsc ($DESCR ("NUMBER"), &_res_temp_descr);
        }
        maxwindow = curline;
        /*
         * Get input from fake field of scrolled line and do what it says:
         *      Kpd . or RETURN/ENTER   => return to menu
         *      UPARROW or TAB          => scroll forward
         *      DOWNARROW or BACKSPACE  => scroll backward
         *      all others              => ignore
         * Note that there is no form function key VAR so this routine
         * handles all terminators itself (by ignoring illegal ones).
         */
        fdv$get (&_fake, &terminator, $DESCR ("FAKE"));
        while ( ! (terminator == FDV$K_FT_NTR  ||  terminator == FDV$K_KP_PER))
        {
                if (terminator == FDV$K_FT_SFW  ||  terminator == FDV$K_FT_SNX)
                        scrfwd ();
                if (terminator == FDV$K_FT_SBK  ||  terminator == FDV$K_FT_SPR)
                        scrbak ();
                fdv$get (&_fake, &terminator, $DESCR ("FAKE"));
        }

}
```

```
/*
 * Subroutine SCRFWD -- Scroll forward.
 *      curline is the line in the resister that the cursor is on.
 *      minwindow and maxwindow delimit the part of the resister
 *      currently displayed in the scrolled area
 */

scrfwd ()
{
        static  struct  dsc$descriptor_s
                _scr_temp_descr = { 0, DSC$K_DTYPE_T, DSC$K_CLASS_S, 0 };

        /*
         * If cursor is at the end of the resister, report, and return
         */
        if (curline == lastresnum)
        {
                fdv$putl ($DESCR ("Last line of resister"));
                return;
        }

        /*
         * If cursor not at the last line of a window, just move down
         * If cursor is  at the last line of a window,
         *      move window forward one line,
         *      write the new last line to the last line of the scrolled area
         * Move current line pointer forward
         */
        if (curline != maxwindow)
                fdv$pft (&FDV$K_FT_SFW, $DESCR ("NUMBER"));
        else
        {
                ++minwindow;
                ++maxwindow;
                LENGTH (_scr_temp_descr) = sizeof resarray [0] - 2;
                POINTER (_scr_temp_descr) = & resarray [maxwindow];
                fdv$pft (&FDV$K_FT_SFW, $DESCR ("NUMBER"), &_scr_temp_descr);
        }

        ++curline;

}
```

```
/*
 * Subroutine SCRBAK -- Scroll backward
 *      curline is the line in the register that the cursor is on.
 *      minwindow and maxwindow delimit the part of the register
 *      currently displayed in the scrolled area
 */

scrbak ()
{
        static  struct  dsc$descriptor_s
                _scr_temp_descr = { 0, DSC$K_DTYPE_T, DSC$K_CLASS_S, 0 };

        /*
         * If the cursor is at the beginning of the register, report, and return
         */
        if (curline == 1)
        {
                fdv$putl ($DESCR ("First line of register"));
                return;
        }

        /*
         * If cursor not at first line of the window, just move up
         * If cursor is  at first line of the window,
         *      move window back one line,
         *      write the new first line to the first line of the scrolled area
         * Move current line pointer back
         */
        if (curline != minwindow)
                fdv$pft (&FDV$K_FT_SBK, $DESCR ("NUMBER"));
        else
        {
                --minwindow;
                --maxwindow;
                LENGTH (_scr_temp_descr) = sizeof regarray [0] - 1;
                POINTER (_scr_temp_descr) = & regarray [minwindow];
                fdv$pft (&FDV$K_FT_SBK, $DESCR ("NUMBER"), &_scr_temp_descr);
        }

        --curline;

}
```

```
/*
 * Subroutine VUEACT
 *      View the account data.
 *      If operator knows the secret word, let operator change
 *      the account data for this session.
 */

vueact ()
{
        static  char    password [sizeof account.opw];
        static  $DESCRIPTORM (_password, password);
        static  struct  dsc$descriptor_s
                _account = { sizeof account - 2, DSC$K_DTYPE_T, DSC$K_CLASS_S, &account };


        fdv$cdisp ($DESCR ("ACCOUNT_DATA"));
        srvchk ();
        fdv$putal (&_account);
        fdv$putd ($DESCR ("SECRET"));

        /*
         * This is not the best way to do protection, just a way of showing
         * another FMS feature. At this point, supervisor mode is on, so the
         * only input allowed is to the password field.
         * If operator doesn't know password, return to menu.
         */
        fdv$getal (0, &terminator);     /* Don't care about value now */
        if (terminator == FDV$K_KP_PER)
                return;
        fdv$ret (&_password, $DESCR ("SECRET"));
        if (strncmp (account.opw, password, sizeof account.opw))
                return;

        /*
         * Allow input from other fields and read from them.
         * If read is terminated by keypad period, don't change account.
         */
        fdv$spoff ();
        simulate_getal ();      /* Read all fields */
        fdv$spon ();            /* Not really needed, just showing off. */
        if (terminator != FDV$K_KP_PER)
```

```
        {
                fdv$retal (&_account);
                fmtchk ();      /* Update the check workspace */
        }

}



/*
 * Simulate action of fdv$setal, using fdv$setaf and PFT.  Could
 * replace this whole routine with a call on fdv$setal, but this shows
 * how mainline program can allow same operator freedom of filling in
 * fields but still regain control after each or changed field.
 * Technique is to read any field, looking only at terminator, then do
 * a process field terminator call to do the operator's action.
 * This technique can be used with calls on fdv$set or fdv$setaf.
 * This example starts with a GET on field '*', first field on form.
 */

simulate_setal ()
{
        static  char    fieldname [6], junk;
        static  $DESCRIPTOR (_fieldname, fieldname);
        static  $DESCRIPTOR1 (_junk, junk);
        static  $DESCRIPTOR (_asterisk, "*");
        int     fieldindex;


        fdv$set (&_junk, &terminator, &_asterisk);
        fdv$retfn (&_fieldname, &fieldindex);   /* Get first field's name */
        while (1)
        {
                /*
                 * Do any special processing for field fieldname at this point.
                 * ...
                 * Go to next or previous field or leave form
                 */
                fdv$pft (&terminator);
```

```
                /*
                 * If status is error, then PFT failed because terminator was
                 * a Keypad Key, which means return to caller.
                 */
                if (fmsstatus < 0)
                        return;
                if (terminator == FDV$K_FT_NTR)
                        if (fmsstatus != 2)
                                return;
                        else
                        {
                                fdv$putl ($DESCR ("INPUT REQUIRED"));
                                fdv$bell ();
                        }

                /*
                 * Go set any other field, returning its name
                 */
                fdv$setaf (&_Junk, &terminator, &_Fieldname, &Fieldindex);
        }

}
```

```
/*
 * Subroutine GETSTA
 * Check FMS status by calling fdv$stat.
 * If not success (>0), print and stop
 */

getsta ()
{
        fdv$stat (&fmsstatus, &rmsstatus);
        if (fmsstatus > 0)
                return;
        error ();       /* and never come back */
}


/*
 * Subroutine SRVCHK
 * Check FMS status by looking at the status recording variables.
 */

srvchk ()
{
        if (fmsstatus > 0)
                return;
        error ();       /* and never come back */
}



/*
 * There is an error returned in the status variables. Detach the
 * terminal to clean up, then print the errors, and stop.
 */

error ()
{
        fdv$dterm (&_tcarea);
        printf ("FDV ERROR.\n  FMS STATUS: %d\n  RMS STATUS: %d",
                fmsstatus, rmsstatus);
        exit (1);
}
```

```
/*
 * VALID1
 *      UAR for field validation of any one character field. The
 *      UAR associated data has in it the legal characters allowed,
 *      except that blank is not allowed unless it appears before
 *      the first trailing blank.  For example an assoc. value string
 *      "aqr" implies that only the letters a, q, and r are allowed.
 *      A string " aqr" means that blank is acceptable in addition
 *      to a, q, and r. Note that this routine is case sensitive
 *      (that is, it checks for correct case).  You can get around
 *      case sensitivity by using the force upper case field attribute
 *      and putting only capitals into the UAR associated value
 *      string.
 *
 *      This routine can be used with any form and field since
 *      it determines the context for itself.
 */

int     valid1 ()
{
        static  int     tca, wksp, curpos, fldtrm, insovr, helpnum;
        static  char    frmnam [31], uarval [81], fldname [31], fvalue;
        static  $DESCRIPTORM (_frmnam, frmnam);
        static  $DESCRIPTOR (_fvalue, fvalue);
        $DESCRIPTOR (_uarval, uarval);
        $DESCRIPTORM (_fldname, fldname);
        int     findex;


        /*
         * Retrieve context: we will ignore tca address, wksp address, frmnam,
         *      curpos, fldtrm, and insovr, using only uarval, and only the
         *      initial, non-blank characters of it.
         * Retrieve field name and index.
         * Retrieve field value.
         */
        fdv$retcx (&tca, &wksp, &_frmnam, &_uarval, &curpos, &fldtrm, &insovr, &helpnum );
        fdv$retfn (&_fldname, &findex);
        fdv$retfn (&_fldname, &findex);
        fdv$ret (&_fvalue, &_fldname, &findex);
```

```
        /*
         * To be valid, fvalue must occur in the string uarval
         */
        trim (&_uarval);
        uarval [LENGTH (_uarval)] = '\0';
        if (strchr (uarval, fvalue) != 0)
                return FDV$K_UVAL_SUC;          /* Success */

        fdv$putl ($DESCR ("Illegal value"));
        return FDV$K_UVAL_FAIL;

}



/*
 * TAKE15
 *      Function Key User Action Routine for the MENU form of SAMP.
 *      Convert Keypad 1-5 into field values 1-5.
 *      Convert Keypad period into field value 1.
 *      Reject all other function keys with error message.
 */

int     take15 ()
{
        static  int     tca, wksp, curpos, fldtrm, insovr, helpnum;
        static  char    frmnam [4], uarval, value;
        static  $DESCRIPTOR (_frmnam, frmnam);
        static  $DESCRIPTOR1 (_uarval, uarval);
        static  $DESCRIPTOR1 (_value, value);


        /*
         * Retrieve context: we will ignore tca address, wksp address, frmnam,
         *      uarval, curpos, and insovr, using only fldtrm
         */
        fdv$retcx (&tca, &wksp, &_frmnam, &_uarval, &curpos, &fldtrm, &insovr, &helpnum );
```

```
/*
 * Do the conversion, displaying the value converted if found.
 * Reject if not one of the expected terminators.
 */
switch (Fldtrm)
{
case FDV$K_KP_PER:
case FDV$K_KP_1:
        value = '1';
        break;
case FDV$K_KP_2:
        value = '2';
        break;
case FDV$K_KP_3:
        value = '3';
        break;
case FDV$K_KP_4:
        value = '4';
        break;
case FDV$K_KP_5:
        value = '5';
        break;
default:
        Fdv$Putl ($DESCR ("Illegal Function Key"));
        Fdv$Sigop ();
        /* Just ignore it now */
        return FDV$K_UKEY_SUC;
}

Fdv$Put (&_value, $DESCR ("OPTION"));
/* Treat as if it is RETURN */
return FDV$K_UKEY_NTR;

}
```

```
/*
 * PASSKY
 *      General function key var to pass only those from the (small) list
 *      in the var associated value string and reject all others.
 *      The list is of the form: n <oneblank> n <oneblank> ... n <manyblanks>
 *      For example the string "110 112" would accept Keypad period and
 *      Keypad zero but no other function keys.
 */

int     passKy ()
{
        static  int     tca, wKsp, curpos, fldtrm, insovr, helpnum;
        static  char    frmnam [4], varval [82];
        static  $DESCRIPTORM (_frmnam, frmnam);
        static  $DESCRIPTORM (_varval, varval);
        char    *nonblank, *nextblank;


        /*
         * Retrieve context: we will ignore tca address, wKsp address, frmnam,
         *      insovr, and curpos, using only fldtrm and varval.
         */
        fdv$retcx (&tca, &wKsp, &_frmnam, &_varval, &curpos, &fldtrm, &insovr, &helpnum );

        /*
         * Break up the list into numbers.  Check each against the actual
         * terminator. If terminator found in list, return success.
         */
        nonblank = varval;      /* Beginning of string */
        while (*nonblank  = ' '  &&  nonblank < & varval [80])
        {
                nextblank = strchr (nonblank, ' ');
                if (fldtrm == val (nonblank, nextblank - nonblank))
                        return FDV$K_UKEY_TRM;  /* Pass Key to application */
                nonblank = nextblank + 1;
        }

        return FDV$K_UKEY_ERR;  /* Let FDV do the beeping */

}
```

```
/*
 * CHKCHK
 *      UAR for SAMP CHECK form.  Makes sure that the check amount is
 *      less than or equal to the current balance.  If not, complain and
 *      change video attributes on balance field so the potential bouncer
 *      can see what there is to work with.
 */

int     chkchk ()
{
        static  char    balance [] = "000000", amtpay [] = "000000" ;
        static  $DESCRIPTOR (_balance, balance);
        static  $DESCRIPTOR (_amtpay, amtpay);
        int     blinkbold;


        fdv$ret (&_balance, $DESCR ("BALANCE"));
        fdv$ret (&_amtpay, $DESCR ("AMTPAY"));
        if (atoi (balance) >= atoi (amtpay))
        {
                blinkbold = -1;         /* Restore to original */
                fdv$afva (&blinkbold, $DESCR ("BALANCE"));
                return FDV$K_UVAL_SUC;
        }

        blinkbold = 3;                  /* Make it very visible */
        fdv$afva (&blinkbold, $DESCR ("BALANCE"));
        fdv$putl ($DESCR ("Your balance doesn't cover that much, reenter amount"));
        return FDV$K_UVAL_FAIL;

}



/*
 * RANGE
 *      General purpose UAR to check the range of any numeric item.  The
 *      associated UAR data must have one of the four forms:
 *              L,U<space>{message}
 *              ,U<space>{message}
```

```
 *                L,<space>{message}
 *                ,<space>{message}
 *        where L is lower bound, U is upper bound, and {message} is an
 *        optional error message in case the field value is out of bounds.
 *        If one of the bounds isn't given, it isn't checked for.  If neither
 *        bound is given, nothing is checked, everything succeeds.  If the
 *        UAR value doesn't have a comma, a FDV$_UAR error message is returned
 *        to the calling program by the FDV so the form designer has to go
 *        back and do it right. If no {message} is given, a simple
 *        "out of range U:L" message is given to the hapless operator.
 *
 *        This UAR can work with any form and numeric field since it gets
 *        context itself.  Care must be taken with fields using field marker
 *        periods since those periods are not returned to the program.
 */

int     range ()
{
        static  int     tca, wksp, curpos, fldtrm, insovr, helpnum;
        static  char    frmnam [31], uarval [81], fldname [31],
                        num_string [133], range_msg [81];
        static  $DESCRIPTORM (_frmnam, frmnam);
        static  $DESCRIPTOR (_range_msg, range_msg);
        $DESCRIPTOR (_uarval, uarval);
        $DESCRIPTORM (_fldname, fldname);
        $DESCRIPTOR (_num_string, num_string);
        char    *blank, *comma;
        int     index, number;


        /*
         * Get context which yields associated data value (ignore other stuff).
         * Get current field name and index.
         * Get field value.
         */
        fdv$retcx (&tca, &wksp, &_frmnam, &_uarval, &curpos, &fldtrm, &insovr, &helpnum );
        fdv$retfn (&_fldname, &index);
        fdv$ret (&_num_string, &_fldname, &index);
        number = atoi (num_string);
```

```
        /*
         * Find comma and blank delimiters.
         * Check for lower bound.
         */
        uarval [LENGTH (_uarval)] = '\0';
        if ((comma = strchr (uarval, ',')) == 0)
                return 0;       /* Illegal uarval string, FDV returns error */

        blank = strchr (comma + 1, ' ');
        if (comma != uarval  &&  number < val (uarval, comma - uarval))
                goto bound_error;

        /*
         * Check for upper bound
         */
        if (blank != comma + 1  &&  number > val (comma + 1, blank - comma - 1))
                goto bound_error;

        /*
         * Passed both tests successfully, return success for UAR value
         */
        return FDV$K_UVAL_SUC;


bound_error:
        /*
         * Error in one of the bounds.
         * Give error message: either from the uarval or make one up.
         */
        if (*(blank + 1) != ' ')
                strcpy (range_msg, blank + 1);
        else
                sprintf (range_msg, "Field value out of bounds.  Must be in range \"%.*s\".",
                         blank - uarval, uarval);

        _ENGTH (_range_msg) = strlen (range_msg);
        fdv$putl (&_range_msg);

        fdv$sigop ();           /* Beep, too. */
        return FDV$K_UVAL_FAIL;

}
```

```
/*
 * The following functions were added for the VAX-11 C version of this program.
 * They are needed to perform some of the string handling functions which are
 * performed in other languages by means of built-in functions.
 */

/*
 * Function VAL
 *      Converts its ASCII string argument to an integer.
 */

static  int     val (string, size)

char    *string;
int     size;

{
        char    temp [16];

        strncpy (temp, string, size);
        temp [size] = '\0';
        return atoi (temp);
}

/*
 * Function ITOA
 *      Converts its int argument to an ASCII string,
 *      and returns a pointer to a string descriptor for it.
 */

static  struct  dsc$descriptor_s        *itoa (i)
int     i;

{
        static  char    string [12];
        static  $DESCRIPTOR (_string, string);

        sprintf (string, "%d", i);
        LENGTH (_string) = strlen (string);

        return &_string;
}
```

```
/*
 * Function TRIM
 *      Removes trailing blanks and tabs from a string passed by descriptor.
 *      (Actually, it just adjusts the length field in the descriptor.)
 */

static  *trim (dp)

struct  dsc$descriptor_s        *dp;

{
        char    *p;

        for (p = POINTER ((*dp)) + LENGTH ((*dp)) - 1;
             p >= POINTER ((*dp));
             --p)
                if (*p != ' '  &&  *p != '\t')
                        break;
        LENGTH ((*dp)) = p - POINTER ((*dp)) + 1;
}
```

```
/* FDVDEF.H
 * Include file for FDV symbols
 */

/*************************/
/* FMS terminator codes: */
/*************************/
#define FDV$K_FT_NTR        0  /*Enter (i.e. end GETs)*/
#define FDV$K_FT_NXT        1  /*Next field */
#define FDV$K_FT_PRV        2  /*Previous field */
#define FDV$K_FT_ATB        3  /*Automatically move to next field*/
#define FDV$K_FT_XBK        4  /*Exit scrolled area backward*/
#define FDV$K_FT_XFW        5  /*Exit scrolled area forward*/
#define FDV$K_FT_SNX        6  /*Scroll forward to next field */
#define FDV$K_FT_SPR        7  /*Scroll backward to previous field*/
#define FDV$K_FT_SFW        8  /*Scroll forward*/
#define FDV$K_FT_SBK        9  /*Scroll backward*/
#define FDV$K_FT_ILG_NXT   11  /*Illegal context for next field */
#define FDV$K_FT_ILG_PRV   12  /*Illegal context for previous field */
#define FDV$K_FT_ILG_ATB   13  /*Illegal context for auto move to next field*/
#define FDV$K_FT_ILG_XBK   14  /*Illegal context for exit scrolled area backward*/
#define FDV$K_FT_ILG_XFW   15  /*Illegal context for exit scrolled area forward*/
#define FDV$K_FT_ILG_SFW   16  /*Illegal context for scroll forward*/
#define FDV$K_FT_ILG_SBK   17  /*Illegal context for scroll backward*/
/*************************************************************/
/* Function Key terminators returned from GETs and WAIT */
/* Also used as FDV keycodes for use with DFKBD.  */
/*************************************************************/
#define FDV$K_AR_UP         99
#define FDV$K_AR_DOWN      100
#define FDV$K_AR_LEFT      101
#define FDV$K_AR_RIGHT     102
#define FDV$K_PF_1         103
#define FDV$K_PF_2         104
#define FDV$K_PF_3         105
#define FDV$K_PF_4         106
#define FDV$K_KP_NTR       107
#define FDV$K_KP_COM       108
#define FDV$K_KP_HYP       109
#define FDV$K_KP_PER       110
#define FDV$K_KP_0         112
#define FDV$K_KP_1         113
```

```
#define FDV$K_KP_2          114
#define FDV$K_KP_3          115
#define FDV$K_KP_4          116
#define FDV$K_KP_5          117
#define FDV$K_KP_6          118
#define FDV$K_KP_7          119
#define FDV$K_KP_8          120
#define FDV$K_KP_9          121
#define FDV$K_GAR_UP        227
#define FDV$K_GAR_DOWN      228
#define FDV$K_GAR_RIGHT     229
#define FDV$K_GAR_LEFT      230
#define FDV$K_GPF_1         231
#define FDV$K_GPF_2         232
#define FDV$K_GPF_3         233
#define FDV$K_GPF_4         234
#define FDV$K_GKP_NTR       235
#define FDV$K_GKP_COM       236
#define FDV$K_GKP_HYP       237
#define FDV$K_GKP_PER       238
#define FDV$K_GKP_0         240
#define FDV$K_GKP_1         241
#define FDV$K_GKP_2         242
#define FDV$K_GKP_3         243
#define FDV$K_GKP_4         244
#define FDV$K_GKP_5         245
#define FDV$K_GKP_6         246
#define FDV$K_GKP_7         247
#define FDV$K_GKP_8         248
#define FDV$K_GKP_9         249
/*************************************************/
/* FDV KeyFunctions. For use in DFKBD call. */
/*************************************************/
#define FDV$K_KF_GOLD         1
#define FDV$K_KF_RESET        2
#define FDV$K_KF_CRSLF        3
#define FDV$K_KF_CRSRT        4
#define FDV$K_KF_DLCHR        5
#define FDV$K_KF_DLFLD        6
#define FDV$K_KF_INS          7
#define FDV$K_KF_OVR          8
#define FDV$K_KF_RFRSH        9
```

```
#define FDV$K_KF_HELP     10
#define FDV$K_KF_NXT      11
#define FDV$K_KF_PRV      12
#define FDV$K_KF_NTR      13
#define FDV$K_KF_SBK      14
#define FDV$K_KF_SFW      15
#define FDV$K_KF_XBK      16
#define FDV$K_KF_XFW      17
#define FDV$K_KF_NONE      0
#define FDV$K_KF_DFLT    (-1)
/*****************************************************************/
/* UAR return codes. These codes are returned  UAR to FDV. */
/*****************************************************************/
/* Field completion return codes */
/*********************************/
#define FDV$K_UVAL_SUC        1000        /*Field completion success      */
#define FDV$K_UVAL_FAIL       1001        /*Field completion failure      */
#define FDV$K_UVAL_END        1002        /*Field completion suc-stop UARs*/
/*************************/
/* Help UAR return codes */
/*************************/
#define FDV$K_UHELP_NO        2000        /*No help given, try next step  */
#define FDV$K_UHELPED         2001        /*Help given, continue sequence */
#define FDV$K_UHELP_ALL       2002        /*Help given, repeat UAR        */
/***********************************/
/* Function Key UAR return codes */
/***********************************/
#define FDV$K_UKEY_ERR        3000        /*Fn Key failure, FDV signals   */
#define FDV$K_UKEY_TRM        3001        /*Fn Key success, normal f.K.   */
#define FDV$K_UKEY_NXT        3002        /*Fn Key succ, treat as NEXT    */
#define FDV$K_UKEY_NTR        3003        /*Fn Key succ, treat as ENTER   */
#define FDV$K_UKEY_SUC        3004        /*Fn Key succ, ignore           */

/*****************************************************************************/
/* FDV status codes returned when FDV$... routines are called as functions. */
/* These codes are VMS status codes and can be signalled.  They correspond  */
/* one-to-one with the FMS status codes retrievable from FDV$STAT.          */
/*****************************************************************************/
#define FDV$_SUC 2719889
#define FDV$_INC 2719897
#define FDV$_MOD 2719905
#define FDV$_IMP 2719922
```

```
#define FDV$_FSP 2719930
#define FDV$_IOL 2719938
#define FDV$_FLB 2719946
#define FDV$_ICH 2719954
#define FDV$_FCH 2719962
#define FDV$_FRM 2719970
#define FDV$_FNM 2719978
#define FDV$_LIN 2719986
#define FDV$_FLD 2719994
#define FDV$_NOF 2720002
#define FDV$_DSP 2720010
#define FDV$_NSC 2720018
#define FDV$_DNM 2720026
#define FDV$_DLN 2720034
#define FDV$_UTR 2720042
#define FDV$_IOR 2720050
#define FDV$_IFN 2720058
#define FDV$_ARG 2720066
#define FDV$_INI 2720074
#define FDV$_STR 2720082
#define FDV$_IVM 2720090
#define FDV$_FVM 2720098
#define FDV$_ITT 2720106
#define FDV$_TCA 2720114
#define FDV$_STA 2720122
#define FDV$_WID 2720130
#define FDV$_NFL 2720138
#define FDV$_IBF 2720146
#define FDV$_NDS 2720154
#define FDV$_UDP 2720162
#define FDV$_UAR 2720170
#define FDV$_UNF 2720178
#define FDV$_CAN 2720194
#define FDV$_KIF 2720202
#define FDV$_KEX 2720210
#define FDV$_KTW 2720218
#define FDV$_KIL 2720226
#define FDV$_TMO 2720234
#define FDV$_LLI 2720242
#define FDV$_VAL 2720250
#define FDV$_IFU 2720258
#define FDV$_SYS 2720266
```

```
/*****************************************************************************/
/* FMS status codes returned when FDV$STAT routine is called.                */
/*****************************************************************************/
/* Success codes. */

#define FDV$K_SUC       1
#define FDV$K_INC       2
#define FDV$K_MOD       3

/* Failure code */

#define FDV$K_IMP       (-2)
#define FDV$K_FSP       (-3)
#define FDV$K_IOL       (-4)
#define FDV$K_FLB       (-5)
#define FDV$K_ICH       (-6)
#define FDV$K_FCH       (-7)
#define FDV$K_FRM       (-8)
#define FDV$K_FNM       (-9)
#define FDV$K_LIN       (-10)
#define FDV$K_FLD       (-11)
#define FDV$K_NOF       (-12)
#define FDV$K_DSP       (-13)
#define FDV$K_NSC       (-14)
#define FDV$K_DNM       (-15)
#define FDV$K_DLN       (-16)
#define FDV$K_UTR       (-17)
#define FDV$K_IOR       (-18)
#define FDV$K_IFN       (-19)
#define FDV$K_ARG       (-20)
#define FDV$K_INI       (-21)
#define FDV$K_STR       (-22)
#define FDV$K_FVM       (-23)
#define FDV$K_IVM       (-24)
#define FDV$K_ITT       (-25)
#define FDV$K_TCA       (-26)
#define FDV$K_STA       (-27)
#define FDV$K_WID       (-28)
#define FDV$K_NFL       (-29)
#define FDV$K_IBF       (-30)
#define FDV$K_NDS       (-31)
#define FDV$K_UDP       (-33)
```

```
#define FDV$K_UAR (-34)
#define FDV$K_UNF (-35)
#define FDV$K_CAN (-39)
#define FDV$K_KIF (-40)
#define FDV$K_KEX (-41)
#define FDV$K_KTW (-42)
#define FDV$K_KIL (-43)
#define FDV$K_TMO (-44)
#define FDV$K_LLI (-45)
#define FDV$K_VAL (-47)
#define FDV$K_IFU (-48)
#define FDV$K_SYS (-49)
```

Chapter 5
Programming FMS Applications in VAX–11 COBOL

The FMS Form Driver processes all Form Driver calls according to VAX–11 standards and language-specific rules. These rules define how arguments are passed to the Form Driver and how values are returned to your program. Language-specific information is briefly presented in this manual. For more detail, refer to the VAX–11 COBOL document set.

Your VAX–11 COBOL application program must comply with the requirements of the VAX–11 COBOL FMS interface. Topics discussed in this chapter include:

- Form Driver Routines
  - Invoking Form Driver Routines as Subroutines
  - Accessing Form Driver Status Codes as Functions
- Argument Passing in FMS
- Null Arguments
- FMS Data Types
  - Character Strings
  - Longword Binary Integers
  - Word Binary Integers
- Non-FMS Data Types
- COBOL Declarations
- One-Dimensional Arrays
- Allocation: Workspace, Terminal Control Area, and Run-Time Memory-Resident Form Area
- Precautions for Using FMS
- Data Conversion
- Sample Application Program in VAX–11 COBOL

A sample program written in COBOL (SAMPCOB.COB) appears at the end of this chapter. Following the code for SAMPCOB.COB are definition files created for the Sample Application. Command file information needed to build the Sample Application program is in Section 5.11.2.

Examples from the Sample Application are used throughout the text to illustrate language issues. Where appropriate examples from SAMPCOB.COB do not exist, other examples are provided.

5.1 Form Driver Routines

You can call any FMS routine as a subroutine or as a function. Syntax follows standard VAX–11 COBOL requirements.

5.1.1 Invoking Form Driver Routines as Subroutines

You use the procedure call statement to invoke an FMS Form Driver routine. For example:

```
CALL "FDV$WAIT".
```

Calls the Form Driver routine FDV$WAIT and passes no arguments.

```
CALL "FDV$GET" USING BY DESCRIPTOR D-MENU-OPTION
                     BY REFERENCE TERMINATOR
                     BY DESCRIPTOR N-MENU-OPTION.
```

Calls the Form Driver routine FDV$GET and passes three arguments.

See Appendix A for a complete list of Form Driver calls. The calling sequence for each Form Driver routine, data access codes, data types, and passing mechanisms are presented in language-independent notation as specified by the VAX–11 Procedure Calling and Condition Handling Standard. For further detail about the VAX–11 Procedure Calling and Condition Handling Standard, refer to the *VAX–11 Run-Time Library Reference Manual.*

5.1.2 Accessing Form Driver Status Codes as Functions

An FMS status code is returned to the calling program at the completion of all Form Driver calls. To receive the returned status code from a Form Driver routine, you use the CALL statement with a GIVING clause. Note that this returns a standard VMS status code. For portability, other status mechanisms can also be used. (For more information, see the *VAX–11 FMS Form Driver Reference Manual*, Chapter 2.)

The following statements call FDV$GET as an FMS function:

```
CALL "FDV$GET" USING BY DESCRIPTOR D-MENU-OPTION
                     BY REFERENCE TERMINATOR
                     BY DESCRIPTOR N-MENU-OPTION

GIVING RETURN_STATUS.
```

5.2 Argument Passing in FMS

The argument passing mechanism refers to the way in which data is passed to a called routine. VAX–11 COBOL has four methods for passing arguments:

- By reference
- By descriptor
- By value
- By content

FMS routines, however, expect arguments to be passed only by reference and by descriptor. (However, null arguments are passed by value. See Section 5.3).

By reference specifies that the storage location of the argument is passed to the routine. FMS expects integers to be passed by reference, which is the COBOL default passing mechanism.

By descriptor specifies that the address of a descriptor data structure is passed to the called routine. FMS expects character strings and arrays to be passed by descriptor. But the COBOL default passing mechanism is by reference. To override the COBOL default, include BY DESCRIPTOR in the CALL statement's USING phrase. This will force the argument list entry to use the descriptor mechanism.

In the following example, the FDV$AWKSP call passes the data items WORKSPACE_SIZE and WORKSPACE to the FDV$AWKSP routine. The integer data item WORKSPACE_SIZE is passed by reference, as expected by FMS. Normally in COBOL a character string such as WORKSPACE would also be passed by reference. However, FMS expects to see a character string passed not as a single address, but as a block of storage passed by descriptor. Thus, the USING BY DESCRIPTOR phrase in the call statement is used to force the use of the descriptor mechanism to pass the character string WORKSPACE.

```
DATA DIVISION.
WORKING-STORAGE SECTION.
01    WORKSPACE           PIC X(12)          GLOBAL.
01    WORKSPACE_SIZE      PIC 9(5)    COMP   GLOBAL   VALUE  2000.

CALL "FDV$AWKSP" USING BY DESCRIPTOR WORKSPACE
                       BY REFERENCE WORKSPACE_SIZE.
```

5.3 Null Arguments

When the call syntax includes optional arguments and you do not wish to specify all of the information, you can use null arguments. Each optional argument can be replaced by a zero to simplify the program. The zero functions as a placeholder for the null argument. Optional arguments to the right of the last required argument can simply be omitted from the call.

In the following example, the FDV$GETAL call passes only the field terminator value:

```
CALL "FDV$GETAL" USING BY VALUE     0,
                       BY REFERENCE TERMINATOR.
```

5.4 FMS Data Types

5.4.1 Character Strings

The character string is one of the general data types used by FMS. USAGE IS DISPLAY is the COBOL default for numeric, alphabetic, and alphanumeric items. Any item with USAGE IS DISPLAY can be used with FMS as a string.

5.4.1.1 Passing Character Strings in FMS — Character strings are passed to Form Driver routines by descriptor (see Section 5.2). For example, the character strings for field value (D-MENU-OPTION) and field name (N-MENU-OPTION) are passed to the FDV$GET routine as follows:

```
CALL "FDV$GET" USING BY DESCRIPTOR  D-MENU-OPTION
                     BY REFERENCE   TERMINATOR
                     BY DESCRIPTOR  N-MENU-OPTION.
```

It is not necessary to define your variables to contain field and form names. There is a way to define a character string that is especially useful for FMS calls requiring form names or field names. For example:

```
CALL "FDV$GET" USING BY DESCRIPTOR  D-MENU-OPTION
                     BY REFERENCE   TERMINATOR
                     BY DESCRIPTOR  "OPTION".
```

Note that the data declarations produced by the FMS/DESCRIPTIONS/DECLARATIONS command do not define the variables for field and form names because they can be defined as above.

5.4.1.2 String Length — You must be certain that your strings are initially declared to be long enough to accommodate your FMS data. One option is to declare your strings to be the exact length of the FMS data to be returned. The information provided by the FMS/DESCRIPTION/DECLARATIONS command allows you to do this easily.

Alternatively, a single string variable can be used in different FMS calls to transfer data to or from several forms and fields. You must declare the string variable to be at least as large as the longest field value string that will be returned to your program. You can use the FMS/DESCRIPTION/BRIEF command to get this information. Use the FDV$RETLE call to return the length of the valid portion of the field value in the string variable. This length can then be used when referencing the data that was entered in the field. A useful application of this is in general

purpose user action routines. The following example can be found in the user action routine RANGE in the Sample Application program:

```
01  FLD_NUMBER              PIC X(132).
01  FLD_LENGTH              PIC 9(9)
COMP.
01  FIELD_NAME              PIC X(10).
01  JUSTIFIED_NUMBER        PIC S9(18).

CALL "FDV$RETLE" USING BY REFERENCE  FLD_LENGTH,
                       BY DESCRIPTOR FIELD_NAME.

IF FLD_LENGTH IS GREATER THAN MAX_NUMERIC_CHARS THEN
   SET RETURN_STATUS TO FAILURE
ELSE
   INSPECT FLD_NUMBER (1:FLD_LENGTH) REPLACING ALL SPACES BY ZERO
   MOVE FLD_NUMBER (1:FLD_LENGTH) TO JUSTIFIED_NUMBER
```

After the execution of the FDV$RETLE call, FLD_LENGTH is equal to the length of the field. It is also equal to the valid portion of the string that is defined by the string descriptor FLD_NUMBER. FLD_LENGTH can now be used with COBOL reference modification to reference the data that was entered in the field. Failure to use (1:FLD_LENGTH) when referencing FLD_NUMBER would result in referencing the entire variable, including any blanks used by the Form Driver to pad the string.

5.4.2 Longword Binary Integers

The longword binary integer is another general data type used by FMS. For example, the FDV$ATERM call passes the longword value for terminal control area size (TERM_CONTROL_AREA_SIZE) and logical I/O channel number (LOGICAL_UNIT_TT):

```
CALL "FDV$ATERM" USING BY DESCRIPTOR  TERM_CONTROL_AREA
                       BY REFERENCE   TERM_CONTROL_AREA_SIZE
                       BY REFERENCE   LOGICAL_UNIT_TT.
```

Numeric arguments must be longword binary integers of type PIC S9(n) COMP where [5<=n<=9]. If you try to pass other numeric types, the calls do not work properly. An exception to this is the FDV$DFKBD call (see next section).

5.4.3 Word Binary Integers

The defkbd argument is a word integer array passed when the FDV$DFKBD routine is invoked. FMS expects arrays to be passed by descriptor.

5.5 Non-FMS Data Types

COBOL data types that are not recognized by FMS can be used in your COBOL application program provided they are not passed to the Form Driver.

5.6 COBOL Declarations

FMS can generate skeleton declarations for the fields in FMS forms. Because the file generated is only a skeleton, you may need to edit the declarations. Create a COBOL library file using the following command:

```
FMS/DESCRIPTION/DECLARATIONS
```

At compile time, request the library file by means of the COPY statement in the data division of your program. (Alternatively, you can use a text editor to add the declaration file to the data division of your program.)

For a detailed description of the command and output associated with the COBOL library file, see the *VAX–11 FMS Utilities Reference Manual.* Note that these declarations are not used in the Sample Application program. The Sample Application program uses the FMS Version 1 equivalent.

5.7 One-Dimensional Arrays

One-dimensional arrays are structures that can be used in FMS for the following arguments:

- tca (terminal control area)
- wksp (workspace)
- mloc (memory location)
- defkbd (define keyboard)

You must provide FMS with storage space for these arguments. You can do that by defining them to be:

- longword integer arrays or character strings for tca, wksp, and mloc
- word integer arrays for defkbd

In the Sample Application program, the tca, wksp, and mloc arguments are passed to several Form Driver routines. These arguments are defined as character strings. You may alternatively define these variables to be integer arrays.

The following declarations establish names and storage for the character string variables TERM_CONTROL_AREA, WORKSPACE, CHECK_WORKSPACE, and MENU_FORM:

```
01     TERM_CONTROL_AREA     PIC X(12).
01     WORKSPACE             PIC X(12).
01     CHECK_WORKSPACE       PIC X(12).
01     MENU_FORM             PIC X(2000).
```

5.8 Allocation: Workspace, Terminal Control Area, and Run-Time Memory-Resident Form Area

The FMS workspace, terminal control area, and run-time memory-resident form area variables are character strings or longword integer arrays. These

variables should be placed in a static storage area of your program. Note that this is not done in the Sample Application program. The sample program's structure protects the workspaces, terminal control areas, and run-time memory-resident form areas implicitly.

The Form Driver needs 12 bytes to associate user information about each workspace and terminal control area. The allocation for a run-time memory-resident form area must be the size of the form. The space for all of these variables is allocated by your application program. Note that FMS uses *only* 12 bytes of space for the workspace and terminal control area allocation. Any allocation greater than 12 bytes is wasted. You declare your workspace, terminal control area, and run-time memory-resident form area only once. FMS remembers their addresses after the addresses have been initially passed to the Form Driver.

For each workspace, the Form Driver also allocates an additional amount of storage space based on your estimate of the amount of memory needed to store your largest form. If your estimate is too small, the Form Driver allocates more space automatically, but performance may be affected. An adequate estimate, however, results in more efficient operation of the Form Driver. You can use the FMS/DIRECTORY/FULL command to find out how much space to allocate.

In the following example from the Sample Application program, a workspace is allocated and the FDV$AWKSP routine is called. When the FDV$AWKSP routine is called, the first argument (WORKSPACE) specifies the area of memory to be used for your workspace. The second argument (WORKSPACE—SIZE) specifies an estimate of the workspace size that you will need to display the largest form in your application.

```
DATA DIVISION.
WORKING-STORAGE SECTION.
01    WORKSPACE          PIC X(12)                    GLOBAL.
01    WORKSPACE_SIZE     PIC 9(5)     COMP            GLOBAL      VALUE 2000.

CALL  "FDV$AWSKP  USING  BY DESCRIPTOR  WORKSPACE
                         BY REFERENCE   WORKSPACE_SIZE.
```

5.9 Precautions for Using FMS

5.9.1 Memory Areas Used Exclusively by FMS

The locations for terminal control area, workspace, and run-time memory-resident form area are used exclusively by FMS. The terminal control area and workspace are attached with the FDV$ATERM and FDV$AWKSP calls and remain allocated until the FDV$DTERM and FDV$DWKSP calls are issued or until the program ends. The run-time memory-resident form area, used in the FDV$READ call, remains allocated until the FDV$DEL call is issued or until the program ends. You never touch the terminal control areas, workspaces, or run-time memory-resident form areas in your program except to pass their addresses to the Form Driver.

5.9.2 Why You Should Declare Certain Variables to Be External

Parameters to the following Form Driver routines should be used with caution:

FDV$ATERM	Attach terminal
FDV$AWKSP	Attach form workspace
FDV$READ	Read form into memory
FDV$SSRV	Specifies status reporting variables

For example, once an FDV$SSRV call is issued, the program variables that contain the FMS status and RMS status become volatile and can change at any call point. As a general rule, you should place the status reporting variables in static storage.

In cases where you need both the FMS and RMS statuses, the FDV$STAT routine can be used. Only the FDV$STAT and FDV$SSRV calls provide RMS status. With the FDV$STAT routine, you do not have to worry about volatility. Note that the above precaution applies only to RMS statuses generated by FMS calls. The COBOL RMS–STS and RMS–STL special registers do not interact in any way with FMS.

The locations for terminal control area, workspace, run-time memory-resident form area, and status reporting variables must all continue to exist while the Form Driver is using them. They must remain allocated until the terminal control area and workspace are detached, until forms in memory location are deleted, and until the status reporting variables are no longer used. The variables can be protected by declaring them to be EXTERNAL; otherwise, the compiler might place them in dynamic storage or reuse their storage.

5.10 Data Conversion

FMS uses only ASCII character strings to display data. All information displayed on the terminal screen, and all information received from the terminal operator, is represented as ASCII field values. Manipulation of numeric data requires conversion of ASCII character strings to numeric data, and conversion of numeric data back to ASCII character strings. In the following discussion of conversion routines, you should assume that the receiving data type can support the largest number that is likely to be generated.

Depending on the data types involved, you may need to impose data conversion operations on your variables. Any item with USAGE IS DISPLAY can be used with FMS as a string. Data items defined to be PIC 9 are actually strings and can be used to make your data conversion less complex. They can be passed to FMS and can be used in arithmetic and logical operations. COBOL provides implicit data conversion in such cases. In cases where you have defined your variables to be PIC X, you must do explicit data conversion through the use of the MOVE statement.

NOTE

Even if you define your data as numeric-display (all 9's), FMS could retain blanks in the field which may cause data conversion errors in your program.

Regardless of how you define your strings, you may need to use the INSPECT statement with the REPLACING ALL SPACES BY ZERO clause as part of any string-to-numeric conversion. Otherwise, you might not get the results you expect. Because the INSPECT statement is only used to replace blanks with zeros, INSPECT is unnecessary if you assign the Zero Fill attribute to the relevant fields. (For more detail on assigning the Zero Fill attribute, see the *VAX–11 FMS Utilities Reference Manual.*)

5.10.1 Data Conversion on PIC X Variables

The following example from the Sample Application shows an explicit data conversion operation on variables defined to be PIC X.

```
DATA DIVISION.
WORKING-STORAGE SECTION.
01     D-REGIST-AMTPAY          PIC X(6).
01     N-CHECK-AMTPAY           PIC X(6)    VALUE IS 'AMTPAY'.
01     TMP_REG_ITEM_PAY_AMT     PIC X(6).
01     NUM_REG_ITEM_PAY_AMT     PIC 9(6)    COMP.
01     CURRENT_BALANCE          PIC 9(6)                GLOBAL.
01     TOTAL_PAYMENTS           PIC 9(6)                GLOBAL.
01     N-CHECK-BALANC           PIC 9(6)                GLOBAL.

CALL "FDV$RET" USING BY DESCRIPTOR  D-REGIST-AMTPAY
                     BY DESCRIPTOR  N-CHECK-AMTPAY.
MOVE D_REGIST_AMTPAY TO TMP_REG_ITEM_PAY_AMT.
INSPECT TMP_REG_ITEM_PAY_AMT REPLACING ALL SPACE BY ZERO.
MOVE TMP_REG_ITEM_PAY_AMT TO NUM_REG_ITEM_PAY_AMT.
SUBTRACT NUM_REG_ITEM_PAY_AMT FROM CURRENT_BALANCE.
ADD NUM_REG_ITEM_PAY_AMT TO TOTAL_PAYMENTS.
CALL "FDV$PUT"
      USING          BY DESCRIPTOR  CURRENT_BALANCE
                     BY DESCRIPTOR  N-CHECK-BALANC.
```

In this example, the COBOL statement MOVE transfers the data stored in the alphanumeric field D-REGIST-AMTPAY to the alphanumeric field TMP_REG_ITEM_PAY_AMT. The INSPECT statement replaces the spaces in the alphanumeric field TMP_REG_ITEM_PAY_AMT by zeros, preparing the data for conversion and transfer to an integer field. The MOVE statement again operates to transfer the data stored in TMP_REG_ITEM_PAY_AMT to the integer field NUM_REG_ITEM_PAY_AMT. Now subtraction and addition operations can be performed on the integer data.

After the data operations have been completed, COBOL displays the value for the balance in a right-justified field N-CHECK-BALANC. The rightmost digit from the program is displayed in the field's rightmost character position. The remaining digits of the character expression are placed to the left of the rightmost digit. If output is longer than the field, FMS truncates

on the left. (The Form Driver displays a data length error message (FDV$_DLN) only if you have set FMS Debug mode.)

For other conversion options, see the general conversion routines in the *VAX–11 Run-Time Library Reference Manual.*

5.10.2 Data Conversion on PIC 9 Variables

The following example from the Sample Application shows COBOL's implicit data conversion on variables defined to be PIC 9. Because the Form Editor assigned the Zero Fill attribute to CURRENT_BALANCE, no INSPECT statement is necessary to replace spaces by zeros.

```
01  CURRENT_BALANCE    PIC 9(6).
01  AMOUNT             PIC 9(6).
 .
 .
 .
CALL "FDV$RET" USING BY DESCRIPTOR CURRENT_BALANCE,
                     BY DESCRIPTOR N-CHECK-BALANC.
CALL "FDV$RET" USING BY DESCRIPTOR AMOUNT,
                     BY DESCRIPTOR N-CHECK-AMTPAY.
INSPECT AMOUNT REPLACING ALL SPACES BY ZERO.

IF CURRENT_BALANCE IS NOT LESS THAN AMOUNT THEN
 .
 .
 .
```

5.11 Sample Application Program in VAX–11 COBOL

The FMS Sample Application program (SAMPCOB.COB) is part of the FMS distribution kit. When FMS is installed, SAMPCOB.COB is placed in the directory FMS$EXAMPLES. Designed to be a demonstration program and learning tool, the Sample Application shows most of the features provided by FMS. The entire Sample Application program appears at the end of this chapter.

5.11.1 Definition Files

The files FDVDEF.LIB, SMPCOBUAR.LIB, and SAMPCOB.LIB are part of the Sample Application program package. When FMS is installed, these files are placed in the directory FMS$EXAMPLES. The FDVDEF.LIB, SMPCOBUAR.LIB, and SAMPCOB.LIB files appear after the Sample Application source code.

5.11.1.1 FDVDEF.LIB — FDVDEF.LIB contains a variety of codes for the Form Driver routines used in the Sample Application program. Although these codes have been created for use in SAMPCOB.COB, they can provide you with a helpful starting point as you create definitions for your own application program. The file FDVDEF.LIB includes:

- FMS terminator codes
- Function key terminators returned from the FDV$GET and FDV$WAIT calls

- Form Driver key functions for use with the FDV$DFKBD call
- User action routine (UAR) return codes, which are returned by the UARs to the Form Driver:

  Field completion UAR return codes

  Help UAR return codes

  Function key UAR return codes
- VMS status codes returned when Form Driver routines are called as functions. These codes can be signaled.
- FMS status codes returned when the FDV$STAT routine is called as a function
- Declarations of Form Driver routines

5.11.1.2 SAMPCOB.LIB — The file SAMPCOB.LIB contains the data declarations specific to the Sample Application program. These data definitions are only useful in the context of the sample program.

5.11.1.3 SMPCOBUAR.LIB — The file SMPCOBUAR.LIB includes declarators for variables and constants used in user action routines. Like the other definition tables (FDVDEF.LIB and SMPCOBUAR.LIB), SMPCOBUAR.LIB is a useful model for creating files for your own program.

5.11.2 Command File for Building the Sample Application Program

The command file for building the Sample Application program includes all the information that you need to compile and link SAMPCOB.COB. When FMS is installed, the command file is placed in the directory FMS$EXAMPLES.

```
$!       S A M P C O B . C O M
$!
$!       Compile and link the COBOL version of the FMS V2 Sample Application
$!
$!       The COBOL source files are:      SAMPCOB.COB
$!                                        FDVDEF.LIB
$!                                        SAMPCOB.LIB
$!                                        SMPCOBUAR.LIB
$!
$!       SMPVECTOR.OBJ and SMPMEMRES.OBJ were produced by the FMS commands:
$!
$!         $ FMS/VECTOR/OUTPUT=SMPVECTOR SAMP.FLB
$!         $ FMS/MEMORY/OUTPUT=SMPMEMRES SAMP.FLB/FORM=(HELP_KEYS,HELP_MENU)
$!
$ COBOL SAMPCOB
$ LINK  SAMPCOB, FMS$EXAMPLES:SMPVECTOR, FMS$EXAMPLES:SMPMEMRES
```

```
IDENTIFICATION DIVISION.
PROGRAM-ID. SAMP.
*****************************************************************************
*    The FMS V2 Sample Application                                          *
*****************************************************************************
ENVIRONMENT DIVISION.
CONFIGURATION SECTION.
INPUT-OUTPUT SECTION.
FILE-CONTROL.
        SELECT OUTPUT-FILE ASSIGN TO "SYS$DISK:".
        SELECT SAMP-FILE ASSIGN TO "SYS$DISK:".
DATA DIVISION.
FILE SECTION.
*
*       Create a sequential file for writing checks. (a new file per check)
FD      OUTPUT-FILE
        GLOBAL
        LABEL RECORDS ARE STANDARD
        VALUE OF ID IS "SAMPCH.DAT".
01      POOL                    PIC X(80).
FD      SAMP-FILE
        GLOBAL
        LABEL RECORDS ARE STANDARD
        VALUE OF ID IS "FMS$EXAMPLES:SAMP.DAT".
01      TEMP_ACCOUNT            PIC X(151).
WORKING-STORAGE SECTION.
01      SAMP_FORM_LIB           PIC X(21)                   VALUE "FMS$EXAMPLES:SAMP.FLB".
01      TERM_CONTROL_AREA       PIC X(12)           GLOBAL.
01      WORKSPACE               PIC X(12)           GLOBAL.
01      CHECK_WORKSPACE         PIC X(12)           GLOBAL.
01      MENU_FORM               PIC X(2000).
01      CHECK_FORM              PIC X(3000).
01      DPOSIT_FORM             PIC X(2000).
01      TOTAL_DEPOSIT           PIC 9(9)            GLOBAL.
01      TOTAL_PAYMENTS          PIC 9(6)            GLOBAL.
01      CURRENT_BALANCE         PIC 9(6)            GLOBAL.
01      MAX_DEPOSIT             PIC 9(6)            GLOBAL  VALUE 999999.
01      MAX_PAYMENT             PIC 9(6)            GLOBAL  VALUE 999999.
01      KEY_PAD_MODE            PIC 9           COMP        VALUE 1.
01      SIGNAL_BELL             PIC 9           COMP        VALUE ZERO.
01      LOGICAL_UNIT            PIC 9(5)        COMP        VALUE 10.
01      LOGICAL_UNIT_TT         PIC 9(5)        COMP        VALUE 2.
01      FMS_STATUS              PIC S9(9)       COMP    GLOBAL.
01      RMS_STATUS              PIC S9(9)       COMP    GLOBAL.
01      TERMINATOR              PIC S9(9)       COMP    GLOBAL.
01      TERM_CONTROL_AREA_SIZE  PIC 9(5)        COMP        VALUE 12.
WORKSPACE-SIZE          PIC 9(5)        COMP    GLOBAL  VALUE 2000.
01      CHECK_WORKSPACE_SIZE    PIC 9(5)        COMP    GLOBAL  VALUE 2000.
01      MENU_FORM_SIZE          PIC 9(5)        COMP    GLOBAL  VALUE 2000.
01      CHECK_FORM_SIZE         PIC 9(5)        COMP    GLOBAL  VALUE 3000.
01      DPOSIT_FORM_SIZE        PIC 9(5)        COMP    GLOBAL  VALUE 2000.
01      UNUSED_TRUE_SIZE        PIC 9(5)        COMP    GLOBAL.
```

```
01      FIELD_INDEX             PIC S9(9)       COMP    GLOBAL.
01      CUR_LINE                PIC 99          COMP    GLOBAL.
01      MIN_WINDOW              PIC 99          COMP    GLOBAL.
01      MAX_WINDOW              PIC 99          COMP    GLOBAL.
*
*       Account Record format of first record in SAMP.DAT
*
01      ACCOUNT                                                 GLOBAL.
        05      ACCT_NUMBER             PIC X(5).
        05                              PIC X(7).
        05      ACCT_NAME.
                10      LAST_NAME       PIC X(20).
                10      FIRST_NAME      PIC X(15).
                10      MIDDLE_NAME     PIC X(15).
        05      ACCT_STREET             PIC X(30).
        05      CITY-STATE-ZIP.
                10      CITY            PIC X(20).
                10      STATE           PIC X(2).
                10      ZIP             PIC X(5).
        05      ACCT_HOME_PHONE         PIC X(10).
        05                              PIC X(10).
        05      ACCT_PASSWORD           PIC X(12).
*
*       Register data format of 2nd thru n records in SAMP.DAT
*
01      REGISTER                                                GLOBAL.
        05      REGISTER_ITEM           OCCURS 30 TIMES.
                10      REG_ITEM_NUMBER         PIC 9(4).
                10      REG_ITEM_DATE           PIC X(7).
                10      REG_ITEM_MEMO_PAY_TO    PIC X(35).
                10      REG_ITEM_DEPOSIT_AMT    PIC 9(6).
                10      REG_ITEM_PAY_AMT        PIC 9(6).
                10      REG_ITEM_BALANCE        PIC 9(6).
                10      FILLER                  PIC X(87).

01      REGISTER_MAX                    PIC 9999        GLOBAL  VALUE 30.
01      FOUND-IN-REGISTER                                       GLOBAL.
        05      LAST_REGISTER_NUM       PIC 9(4).
        05      LAST_CHECK_NUM          PIC 9(4).
        05      ACCT_BALANCE            PIC 9(6).
*
*       Deposit data (READ via FDV$GETAL)
*
01      DEPOSIT                                                 GLOBAL.
        05      DEPOSIT_DATE            PIC X(7).
        05      DEPOSIT_CUR_BAL         PIC 9(6).
        05      DEPOSIT_AMOUNT          PIC 9(6).
        05      DEPOSIT_NEW_BAL         PIC 9(6).
        05      DEPOSIT_MEMO            PIC X(35).
*
*       FMS SYMBOLS
*
```

```
COPY "FDVDEF".
*
*       FORM DESCRIPTION STARTS HERE
* NOTE:  The extract from the forms library has been modified to include
*       the GLOBAL declaration.
*
COPY "SAMPCOB".
*
/
PROCEDURE DIVISION.
0.
*+
* Initialize FMS
*   Attach default terminal
*   Attach normal and check workspaces (order important for help
*       and refresh during CHECK/CHKDON time -- try switching and see)
*   Open form library, attach to channel 1
*   Set keypad mode to application
*   Set signal mode to bell (default, but it's fun to do)
*-
        CALL "FDV$ATERM" USING  BY DESCRIPTOR   TERM_CONTROL_AREA
                                BY REFERENCE    TERM_CONTROL_AREA_SIZE
                                BY REFERENCE    LOGICAL_UNIT_TT.
        CALL "GETSTAT".
        CALL "FDV$AWKSP" USING  BY DESCRIPTOR   CHECK_WORKSPACE
                                BY REFERENCE    CHECK_WORKSPACE_SIZE.
        CALL "GETSTAT".
        CALL "FDV$AWKSP" USING  BY DESCRIPTOR   WORKSPACE
                                BY REFERENCE    WORKSPACE_SIZE.
        CALL "GETSTAT".
        CALL "FDV$LOPEN" USING BY DESCRIPTOR    SAMP_FORM_LIB
                                BY REFERENCE    LOGICAL_UNIT.
        CALL "GETSTAT".
        CALL "FDV$SPADA" USING  BY REFERENCE    KEY_PAD_MODE.
        CALL "GETSTAT".
        CALL "FDV$SSIGQ" USING  BY REFERENCE    SIGNAL_BELL.
        CALL "GETSTAT".
*+
* Set all future calls to return status to the two status recording
* variables FMS_STATUS and RMS_STATUS without having to call the
* the FDV$STAT routine.
*-
        CALL "FDV$SSRV" USING   BY REFERENCE    FMS_STATUS
                                BY REFERENCE    RMS_STATUS.
*+
* Read in a few forms from the form library onto the dynamic
* resident form list.  You may be able to detect the difference
* in the form to form access times for those forms which have to be
* accessed from the form library on disk and those forms which are
* on the dynamic or static memory resident form list.  See the
* installation notes for this program (the LINK command) to see
* which forms are on the static memory resident form list.
*-
```

```
        CALL "FDV$READ" USING    BY DESCRIPTOR   FORM-MENU
                                 BY DESCRIPTOR   MENU_FORM
                                 BY REFERENCE    MENU_FORM_SIZE
                                 BY REFERENCE    UNUSED_TRUE_SIZE.

        CALL "FDV$READ" USING    BY DESCRIPTOR   FORM-CHECK
                                 BY DESCRIPTOR   CHECK_FORM
                                 BY REFERENCE    CHECK_FORM_SIZE
                                 BY REFERENCE    UNUSED_TRUE_SIZE.

        CALL "FDV$READ" USING    BY DESCRIPTOR   FORM-DPOSIT
                                 BY DESCRIPTOR   DPOSIT_FORM
                                 BY REFERENCE    DPOSIT_FORM_SIZE
                                 BY REFERENCE    UNUSED_TRUE_SIZE.

*+
* Initialize account information
*-
        CALL "INACCT".
*+
* Put up welcome form, Wait for response
*-
        CALL "FDV$CDISP" USING  BY DESCRIPTOR   FORM-WELCOM.
        CALL "GETSTAT".
        CALL "FDV$WAIT".
*+
* Process all menu requests
*-
        CALL "MENU".
*+
* Clean up and leave:
*   Close form library.
*   Reset Keypad to numeric.
*   Delete a form from dynamic mem. res. form list just to show how.
*   Detach workspaces (not really necessary since DTERM would do it).
*   Detach terminal.
*   Delete a form from dynamic mem. res. form list just to show how.
*-
        CALL "FDV$LCLOS".
        MOVE ZERO TO KEY-PAD-MODE.
        CALL "FDV$SPADA" USING  BY REFERENCE    KEY_PAD_MODE.
        CALL "FDV$DEL"  USING   BY DESCRIPTOR   FORM-MENU.
        CALL "FDV$DWKSP" USING  BY DESCRIPTOR   WORKSPACE.
        CALL "GETSTAT".
        CALL "FDV$DWKSP" USING  BY DESCRIPTOR   CHECK_WORKSPACE.
        CALL "GETSTAT".
        CALL "FDV$DTERM" USING  BY DESCRIPTOR   TERM_CONTROL_AREA.
        EXIT PROGRAM.
```

```
IDENTIFICATION DIVISION.
PROGRAM-ID.     INACCT.
*+
*       Read from file SAMP.DAT into internal variables.
*       Set up the workspace for checks and fill in the check form
*       with the account's name, address, and account number.
*-
DATA DIVISION.
WORKING-STORAGE SECTION.
01      EOF-FLAG                PIC S9(9)       COMP.
PROCEDURE DIVISION.
0.
*+
* Open file, set account data.  The first record in the file is the
* account data. The 2nd thru n records are the check register data.
* The last record has the current balance data.
*-
        OPEN INPUT SAMP-FILE.
        READ SAMP-FILE INTO ACCOUNT
                AT END  DISPLAY "Error on SAMP.DAT"
                        STOP RUN.
*+
* Read the remaining records into the check register, counting them.
* The last record has the current balance, and some record has the
* last check number used (not necessarily the last record).
*-
        MOVE ZERO TO LAST_CHECK_NUM.
        SET EOF-FLAG TO FAILURE.
        PERFORM WITH TEST AFTER VARYING LAST_REGISTER_NUM FROM 1 BY 1 UNTIL
            EOF-FLAG IS SUCCESS OR LAST_REGISTER_NUM NOT < REGISTER_MAX
                MOVE SPACES TO TEMP_ACCOUNT
                READ SAMP-FILE NEXT RECORD INTO REGISTER_ITEM(LAST_REGISTER_NUM)
                        AT END  SET EOF-FLAG TO SUCCESS END-READ
                IF EOF-FLAG IS FAILURE THEN
                        INSPECT REG_ITEM_NUMBER(LAST_REGISTER_NUM) REPLACING ALL SPACE BY ZERO
                        INSPECT REG_ITEM_DEPOSIT_AMT(LAST_REGISTER_NUM) REPLACING ALL SPACE BY ZERO
                        INSPECT REG_ITEM_PAY_AMT(LAST_REGISTER_NUM) REPLACING ALL SPACE BY ZERO
                        INSPECT REG_ITEM_BALANCE(LAST_REGISTER_NUM) REPLACING ALL SPACE BY ZERO
                        IF REG_ITEM_NUMBER(LAST_REGISTER_NUM) NOT EQUAL ZERO THEN
                                MOVE REG_ITEM_NUMBER(LAST_REGISTER_NUM) TO LAST_CHECK_NUM END-IF
                        IF REG_ITEM_BALANCE(LAST_REGISTER_NUM) NOT EQUAL ZERO THEN
                                MOVE REG_ITEM_BALANCE(LAST_REGISTER_NUM)
                                        TO ACCT_BALANCE, CURRENT_BALANCE END-IF
                END-IF
        END-PERFORM.
        SUBTRACT 1 FROM LAST_REGISTER_NUM.
```

```
*+
* Check for data file in error.
* Take balance from last record read.
* Set session sums to zero to say no activity yet.
*-
        EVALUATE         TRUE
                WHEN    LAST_REGISTER_NUM = 1    STOP "SAMP.DAT data error on last_register_num"
                WHEN    LAST_CHECK_NUM    = ZERO STOP "SAMP.DAT data error on last_check_num"
                WHEN    ACCT_BALANCE      = ZERO STOP "SAMP.DAT data error on ACCT_BALANCE"
                WHEN    CURRENT_BALANCE   = ZERO STOP "SAMP.DAT data error on CURRENT_BALANCE"
        END-EVALUATE.
*+
* Set up the check workspace once so we don't have to do it every time.
*-
*
        CALL "FMTCHK".
        CLOSE SAMP-FILE.
        EXIT PROGRAM.
END PROGRAM INACCT.



IDENTIFICATION DIVISION.
PROGRAM-ID.     MENU.
*+
*       Accept inputs from the menu form and dispatch to the
*       appropriate routine.  Repeat until option 1 (exit) is
*       chosen.  The UARs in the form guarantee that we get back
*       only inputs '1'-'5' with the correct terminators.
*       Options are:
*               1 => Exit
*               2 => Write checks
*               3 => Make deposit
*               4 => View register
*               5 => View account data
*-
*
DATA DIVISION.
WORKING-STORAGE SECTION.
PROCEDURE DIVISION.
0.
        CALL "FDV$CDISP" USING  BY DESCRIPTOR   FORM-MENU.
        MOVE "2" TO D-MENU-OPTION.
        CALL "FDV$PUT" USING    BY DESCRIPTOR   D-MENU-OPTION
                                BY DESCRIPTOR   N-MENU-OPTION.
        CALL "FDV$GET" USING    BY DESCRIPTOR   D-MENU-OPTION
                                BY REFERENCE    TERMINATOR
                                BY DESCRIPTOR   N-MENU-OPTION.
        CALL "SRVCHK".
```

```
        EVALUATE D-MENU-OPTION
                WHEN  "1"       GO TO FINI
                WHEN  "2"       CALL "WRITCH"
                WHEN  "3"       CALL "MAKDEP"
                WHEN  "4"       CALL "VUEREG"
                WHEN  "5"       CALL "VUEACT"
        END-EVALUATE.
        GO TO 0.
FINI.
        EXIT PROGRAM.



IDENTIFICATION DIVISION.
PROGRAM-ID.     MAKDEP.
*+
*       Make a deposit, enter into check register
*       Cancel on Keypad period.
*       Note that the form function key UAR allows only kpd period.
*+
DATA DIVISION.
WORKING-STORAGE SECTION.
01      REG_FULL_MSG            PIC X(35)               VALUE "Register full, can't enter deposit".
01      OVERFLOW_MSG            PIC X(35)               VALUE "Overflow, only 6 digits allowed.".
01      TMP                     PIC X(80)               VALUE SPACE.
01      LARGE_TMP               PIC 9(9)        COMP.
01      BANK_SHARE              PIC 9(7)                VALUE 1000000.
01      FORM-DONE               PIC X(4)                VALUE "DONE".
PROCEDURE DIVISION.
0.
*+
* Put up deposit form with current balance
*-
        CALL "FDV$CDISP" USING  BY DESCRIPTOR   FORM-DPOSIT.
        CALL "SRVCHK".
        CALL "FDV$PUT"  USING   BY DESCRIPTOR   CURRENT_BALANCE
                                BY DESCRIPTOR   N-DPOSIT-CURBAL.
*+
* Get deposit amount and memo from operator.
* Abort on kpd period.
*-
        CALL "FDV$GETAL" USING  BY DESCRIPTOR   DEPOSIT
                                BY REFERENCE    TERMINATOR.
        IF TERMINATOR = FDV$K_KP_PER THEN       GO TO FINI.
*+
* Have deposit information now.
* If no room in check register must abort.
*-
```

```
        IF LAST_REGISTER_NUM NOT LESS THAN REGISTER_MAX THEN
                CALL "FDV$PUTL" USING   BY DESCRIPTOR   REG_FULL_MSG
                        CALL "FDV$WAIT"
                        GO TO FINI.
*+
* Add to balance and session sum.
* Check for overflow (program and form keep only six digits).
* Display new balance.
* Make entry in register.
*-
        INSPECT DEPOSIT_AMOUNT REPLACING ALL SPACE BY ZERO.
        ADD DEPOSIT_AMOUNT TO TOTAL_DEPOSIT.
        ADD DEPOSIT_AMOUNT TO CURRENT_BALANCE ON SIZE ERROR
                CALL "FDV$PUTL" USING   BY DESCRIPTOR   OVERFLOW_MSG
                ADD DEPOSIT_AMOUNT CURRENT_BALANCE GIVING LARGE_TMP
                SUBTRACT BANK_SHARE FROM LARGE_TMP GIVING CURRENT_BALANCE
                CALL "FDV$WAIT".
        CALL "FDV$PUT"  USING   BY DESCRIPTOR   CURRENT_BALANCE
                                BY DESCRIPTOR   N-DPOSIT-NEWBAL.
        ADD 1 TO LAST_REGISTER_NUM.
        MOVE ZEROS TO REG_ITEM_NUMBER(LAST_REGISTER_NUM), REG_ITEM_PAY_AMT(LAST_REGISTER_NUM).
        MOVE DEPOSIT_DATE TO REG_ITEM_DATE(LAST_REGISTER_NUM).
        MOVE DEPOSIT_AMOUNT TO REG_ITEM_DEPOSIT_AMT(LAST_REGISTER_NUM).
        MOVE DEPOSIT_CUR_BAL TO REG_ITEM_BALANCE(LAST_REGISTER_NUM).
        MOVE DEPOSIT_MEMO TO REG_ITEM_MEMO_PAY_TO(LAST_REGISTER_NUM).
        CALL "FDV$RET"  USING   BY DESCRIPTOR   REG_ITEM_BALANCE(LAST_REGISTER_NUM)
                                BY DESCRIPTOR   N-DPOSIT-NEWBAL.
*+
* Sample of how to keep message texts stored with the form rather
* than in a program.  This is especially useful for multi-lingual
* environments: only the form text and the form named data must
* be changed and nothing in the program.  The trick is to store the
* response text in named data. This is the only example of how to do
* it in this program, but all messages could be stored like this.
* Message intent is: "Deposit made, press RETURN or ENTER to continue."
*-
        CALL "FDV$RETDN" USING  BY DESCRIPTOR   FORM-DONE
                                BY DESCRIPTOR   TMP.
        CALL "FDV$PUTL" USING   BY DESCRIPTOR   TMP.
        CALL "FDV$WAIT".
FINI.
        EXIT PROGRAM.
END PROGRAM MAKDEP.
```

```
IDENTIFICATION DIVISION.
PROGRAM-ID.     WRITCH.
*+
*       Write one or more checks
*-
DATA DIVISION.
WORKING-STORAGE SECTION.
01      LED-NUMBER-3            PIC 9           COMP            VALUE 3.
PROCEDURE DIVISION.
0.
*+
* Turn on LED 3 on the VT100 during this routine, just to show how.
*-
        CALL "FDV$LEDON" USING  BY REFERENCE    LED-NUMBER-3.
*+
* Mark WORKSPACE not displayed so it doesn't show up during a refresh.
* Put up CHECK form from already loaded workspace
* and display current balance
*-
        CALL "FDV$NDISP"
        CALL "FDV$SWKSP" USING  BY DESCRIPTOR   CHECK_WORKSPACE.
        CALL "FDV$DISPW".
        CALL "FDV$PUT" USING    BY DESCRIPTOR   CURRENT_BALANCE
                                BY DESCRIPTOR   N-CHECK-BALANC.
*+
* Process checks until a Keypad period is read
*-
        PERFORM WITH TEST AFTER UNTIL TERMINATOR = FDV$K_KP_PER
                CALL "ONECHK"
                CALL "ENDCHK"
        END-PERFORM.
*+
* Turn off LED 3 on VT100
*-
*
        CALL "FDV$LEDOF" USING  BY REFERENCE    LED-NUMBER-3.
        CALL "FDV$SWKSP" USING  BY DESCRIPTOR   WORKSPACE.
        EXIT PROGRAM.



IDENTIFICATION DIVISION.
PROGRAM-ID.     ONECHK.
*+
*       If input is terminated by Kpd period, return with no action
*       Else deduct from balance and enter into resister.
*       Note that a UAR in the form guarantees that the amount of
*       the check is always less than or equal to the balance.
*       Note that the form function key UAR  allows only Kpd period
*       as terminator (other than FDV$K_FT_NTR).
*-
*
```

```
DATA DIVISION.
WORKING-STORAGE SECTION.
01      REG_FULL_MSG            PIC X(35)               VALUE "Register full, cant't enter check.".
01      TMP                     PIC X(80)               VALUE SPACE.
01      TMP_REG_ITEM_PAY_AMT    PIC X(6).
01      NUM_REG_ITEM_PAY_AMT    PIC 9(6)        COMP.
01      NEW_CHECK_NUMBER        PIC 9(4)                VALUE ZERO.
01      NEW_LAST_REGIST_NUM     PIC 9(4)                VALUE ZERO.
PROCEDURE DIVISION.
0.
        ADD 1, LAST_CHECK_NUM    GIVING NEW_CHECK_NUMBER.
        ADD 1, LAST_REGISTER_NUM GIVING NEW_LAST_REGIST_NUM.
        CALL "FDV$PUT" USING    BY DESCRIPTOR   NEW_CHECK_NUMBER
                                BY DESCRIPTOR   N-CHECK-NUMBER.
        CALL "FDV$GETAL" USING  BY DESCRIPTOR   TMP
                                BY REFERENCE    TERMINATOR.
        IF TERMINATOR = FDV$K_KP_PER THEN
                GO TO FINI.
*+
* If the check wouldn't fit in the register, don't process, just
* give error message, wait for acknowledgement, and return
*-
        IF LAST_REGISTER_NUM NOT LESS THAN REGISTER_MAX THEN
                CALL "FDV$PUTL" USING   BY DESCRIPTOR   REG_FULL_MSG
                CALL "FDV$WAIT"
                GO TO FINI.
*+
* Get amount from check.
* Update balance (in memory and on screen) and session sums.
* Transfer form values to register item.
*-
        CALL "FDV$RET"
                USING   BY DESCRIPTOR   D-REGIST-AMTPAY
                        BY DESCRIPTOR   N-CHECK-AMTPAY.
        MOVE D-REGIST-AMTPAY TO TMP_REG_ITEM_PAY_AMT.
        INSPECT TMP_REG_ITEM_PAY_AMT REPLACING ALL SPACE BY ZERO.
        MOVE TMP_REG_ITEM_PAY_AMT TO NUM_REG_ITEM_PAY_AMT.
        SUBTRACT NUM_REG_ITEM_PAY_AMT FROM CURRENT_BALANCE.
        ADD NUM_REG_ITEM_PAY_AMT TO TOTAL_PAYMENTS.
        CALL "FDV$PUT"
                USING   BY DESCRIPTOR   CURRENT_BALANCE
                        BY DESCRIPTOR   N-CHECK-BALANC.

* Now collect the register data and then update the register. AMTPAY has
* already been moved by the previous FDV$RET.

        MOVE NEW_CHECK_NUMBER TO D-REGIST-NUMBER.
        CALL "FDV$RET"
                USING   BY DESCRIPTOR   D-REGIST-DATE
                        BY DESCRIPTOR   N-CHECK-DATE.
```

```
        CALL "FDV$RET"
                USING   BY DESCRIPTOR   D-REGIST-PAYMEM
                        BY DESCRIPTOR   N-CHECK-PAYTO.
        CALL "FDV$RET"
                USING   BY DESCRIPTOR   D-REGIST-BALANC
                        BY DESCRIPTOR   N-CHECK-BALANC.
        MOVE ZERO TO D-REGIST-DPOSIT.
        MOVE SPACE TO D-REGIST-FAKE.
        MOVE SPACE TO D-REGIST-SUMARY.
*+
* Update register array and counters
*-
        MOVE NEW_LAST_REGIST_NUM TO LAST_REGISTER_NUM.
        MOVE NEW_CHECK_NUMBER TO LAST_CHECK_NUM.
        MOVE D-REGIST TO REGISTER_ITEM(LAST_REGISTER_NUM).
FINI.
        EXIT PROGRAM.
END PROGRAM ONECHK.




IDENTIFICATION DIVISION.
PROGRAM-ID.     ENDCHK.
*+
*       Finish off check processing by giving operator
*       three options:
*               RETURN  Write another check
*               KPD 0   Print the check into file SAMPCH.DAT
*               KPD .   Return to menu
*-
DATA DIVISION.
WORKING-STORAGE SECTION.
01      START_LINE              PIC 99      COMP        VALUE 20.
01      LINE_COUNT              PIC 99      COMP        VALUE 4.
PROCEDURE DIVISION.
0.
*       Check to see if check write was aborted by kpd per.
*       If so, then don't give any further choice, just abort.
*       Note that form function key UAR allows only the above
*       terminators to get through.
*-
        IF TERMINATOR NOT = FDV$K_KP_PER THEN
*+
* Tell the operator that the check has been paid by overlaying with
* a new form, using the normal workspace, thereby saving the check
* workspace in case another check is to be written.
*-
                CALL "FDV$SWKSP" USING  BY DESCRIPTOR   WORKSPACE
                CALL "FDV$DISP" USING   BY DESCRIPTOR   FORM_CHKDON
```

```
*+
* Wait for operator to enter either KPD period, NTR, or KPD zero.
* Print the check as many times as requested.
* (Note that a UAR on the form guarantees that only those terminators
* are accepted).
* Process accordingly.
*-
                CALL "FDV$WAIT" USING   BY REFERENCE    TERMINATOR
                PERFORM WITH TEST BEFORE UNTIL TERMINATOR NOT = FDV$K_KP_0
                        CALL "PRICHK"
                        CALL "FDV$WAIT" USING   BY REFERENCE    TERMINATOR
                END-PERFORM
*+
* If choice is to quit,
* then mark check wksp undisplayed so it doesn't appear during refresh,
* else mark normal workspace (occupied by CHKDON form) undisplayed
*   so it doesn't show during refresh and then clear its lines.
*   (Clearing the space occupied by the CHKDON form, lines 20-23
*   is better done by overlaying with a blank form to
*   avoid having to know the lines numbers to clear).
*-
                IF TERMINATOR = FDV$K_KP_PER THEN
                        CALL "FDV$SWKSP" USING  BY DESCRIPTOR   CHECK_WORKSPACE
                        CALL "FDV$NDISP"
                ELSE
                        CALL "FDV$NDISP"
                        CALL "FDV$CLEAR" USING  BY REFERENCE    START_LINE
                                                BY REFERENCE    LINE_COUNT
                        CALL "FDV$SWKSP" USING  BY DESCRIPTOR   CHECK_WORKSPACE
                END-IF
*+
* Going to write another check now or eventually, so:
* Clear out operator entered fields.
*-
                CALL "FDV$PUTD" USING   BY DESCRIPTOR   N-CHECK-AMTPAY
                CALL "FDV$PUTD" USING   BY DESCRIPTOR   N-CHECK-MEMO
                CALL "FDV$PUTD" USING   BY DESCRIPTOR   N-CHECK-PAYTO
        END-IF.
        EXIT PROGRAM.



IDENTIFICATION DIVISION.
PROGRAM-ID.     PRICHK.
*+
*       Print the check into the file SAMPCH.DAT
*       Use the check workspace, then switch back to the normal wksp
*       to keep things clean.
*-
```

```
DATA DIVISION.
WORKING-STORAGE SECTION.
01      CHK_WRITTEN_MSG         PIC X(35)               VALUE "Check written to file.".
01      TMP_FIELD_NAME          PIC X(6).
01      TMP                     PIC XX.
01      FIRSTL                  PIC 99.
01      LASTL                   PIC 99.
01      LINE_NUMBER             PIC 99   COMP.
01      LINE_LENGTH             PIC 999.
PROCEDURE DIVISION.
0.
*+
* Open check writing file.  Note there's a new version for every check.
* Switch workspaces
*+
        OPEN OUTPUT OUTPUT-FILE.
        CALL "FDV$SWKSP" USING  BY DESCRIPTOR   CHECK_WORKSPACE.
*+
* Get the top and bottom lines of the check from the named data
* (first two characters).
*-
        MOVE "FIRST" TO TMP_FIELD_NAME.
        CALL "FDV$RETDN" USING  BY DESCRIPTOR   TMP_FIELD_NAME
                                BY DESCRIPTOR   TMP.
        INSPECT TMP REPLACING ALL SPACE BY ZERO.
        MOVE TMP TO FIRSTL.
        MOVE "LAST" TO TMP_FIELD_NAME.
        CALL "FDV$RETDN" USING  BY DESCRIPTOR   TMP_FIELD_NAME
                                BY DESCRIPTOR   TMP.
        INSPECT TMP REPLACING ALL SPACE BY ZERO.
        MOVE TMP TO LASTL.
*+
* Get lines from form.
* Convert to line printer style.
* Write to file.
*-
        PERFORM VARYING LINE_NUMBER FROM FIRSTL BY 1 UNTIL LINE_NUMBER . LASTL
                MOVE SPACES TO POOL
                CALL "FDV$RETFL" USING  BY REFERENCE    LINE_NUMBER
                                        BY DESCRIPTOR   POOL
                                        BY REFERENCE    LINE_LENGTH
                WRITE POOL
        END-PERFORM.
        CALL "FDV$PUTL" USING   BY DESCRIPTOR   CHK_WRITTEN_MSG.
        CLOSE OUTPUT-FILE.
        CALL "FDV$SWKSP"        USING   BY DESCRIPTOR   WORKSPACE.
        EXIT PROGRAM.
END PROGRAM PRICHK.
END PROGRAM ENDCHK.
END PROGRAM WRITCH.
```

```
IDENTIFICATION DIVISION.
PROGRAM-ID.     VUEACT.
*+
*       View the account data.
*       If operator knows the secret word, let operator change
*       the account data for this session.
*-
DATA DIVISION.
WORKING-STORAGE SECTION.
PROCEDURE DIVISION.
0.
        CALL "FDV$CDISP" USING  BY DESCRIPTOR   FORM-ACTDAT.
        CALL "SRVCHK".
        CALL "FDV$PUTAL" USING  BY DESCRIPTOR   ACCOUNT.
        CALL "SRVCHK".
        CALL "FDV$PUTD" USING   BY DESCRIPTOR   N-ACTDAT-SECRET.
*+
* This is not the best way to do protection, just a way of showing
* another FMS feature. At this point, supervisor mode is on, so the
* only input allowed is to the password field.
* If operator doesn't know the password, return to menu.
*-
        CALL "FDV$GETAL" USING  BY VALUE        0,
                                BY REFERENCE    TERMINATOR.

        IF TERMINATOR IS NOT = FDV$K_KP_PER
        THEN    CALL "FDV$RET"  USING   BY DESCRIPTOR   D-ACTDAT-SECRET
                                        BY DESCRIPTOR   N-ACTDAT-SECRET
                IF D-ACTDAT-SECRET  = ACCT_PASSWORD
*+
* Allow input from other fields and read from them.
* If read is terminated by keypad period, don't change account.
*-
                THEN    CALL "FDV$SPOFF"
                        CALL "READAL"
                        CALL "FDV$SPON"
                        IF TERMINATOR NOT = FDV$K_KP_PER
                        THEN    CALL "FDV$RETAL" USING  BY DESCRIPTOR   ACCOUNT
                                CALL "FMTCHK"
                        END-IF
                END-IF
        END-IF.
        EXIT PROGRAM.



IDENTIFICATION DIVISION.
PROGRAM-ID.     READAL.
*+
* Simulate action of FDV$GETAL, using FDV$GETAF and PFT.  Could
* replace this whole routine with a call on FDV$GETAL, but this shows
```

```
* how mainline program can allow same operator freedom of filling in
* fields but still regain control after each or changed field.
* Technique is to read any field, looking only at terminator, then do
* a process field terminator call to do the operator's action.
* This technique can be used with calls on FDV$GET or FDV$GETAF.
* This example starts with a GET on field '*', first field on form.
*-
DATA DIVISION.
WORKING-STORAGE SECTION.
01      TMP                     PIC X(16)               VALUE SPACE.
01      STAR                    PIC X                   VALUE "*".
01      INPUT_REQ_MSG           PIC X(14)               VALUE "Input Required".
01      FIELD_NAME              PIC X(10)               VALUE SPACE.
01      ESCAPE                  PIC S9(9)       COMP.
PROCEDURE DIVISION.
0.
        CALL "FDV$GET" USING    BY DESCRIPTOR   TMP
                                BY REFERENCE    TERMINATOR
                                BY DESCRIPTOR   STAR.
        CALL "FDV$RETFN" USING  BY DESCRIPTOR   FIELD_NAME
                                BY REFERENCE    FIELD_INDEX.
*+
* If status is error, then PF~ failed because terminator was
* a Keypad Key, which means return to caller.
*-
        SET ESCAPE TO FAILURE.
        PERFORM UNTIL ESCAPE IS SUCCESS
                CALL "FDV$PFT" USING    BY REFERENCE    TERMINATOR
                IF TERMINATOR = FDV$K_KP_PER
                THEN    SET ESCAPE TO SUCCESS
                ELSE    IF FMS_STATUS > ZERO
                        THEN    SET ESCAPE TO SUCCESS
                        ELSE
                                EVALUATE        TERMINATOR      FMS_STATUS
                                WHEN            FDV$K_FT_NTR    NOT FDV$K_INC   SET ESCAPE TO SUCCESS
                                WHEN            FDV$K_FT_NTR    FDV$K_INC               CALL "FDV$PUTL"
                                                                                      USING     BY DESCRIPTOR INPUT_REQ_MSG
                                                                                CALL "FDV$BELL"
                                                                                CALL "FDV$GETAF"
                                                                                      USING     BY DESCRIPTOR   TMP
                                                                                                BY REFERENCE    TERMINATOR
                                                                                                BY DESCRIPTOR   FIELD_NAME
                                                                                                BY REFERENCE    FIELD_INDEX
                                WHEN            NOT FDV$K_FT_NTR            ANY           CALL "FDV$GETAF"
                                                                                      USING     BY DESCRIPTOR   TMP
                                                                                                BY REFERENCE    TERMINATOR
                                                                                                BY DESCRIPTOR   FIELD_NAME
                                                                                                BY REFERENCE    FIELD_INDEX
                        END-EVALUATE
                        END-IF
                END-IF
        END-PERFORM.
        EXIT PROGRAM.
END PROGRAM READAL.
```

```
IDENTIFICATION DIVISION.
PROGRAM-ID.     VUEREG.
*+
*       View the check register and scroll through it.
*       Also display totals for current session.
*
DATA DIVISION.
WORKING-STORAGE SECTION.
01      OVERFLOW_MSG            PIC X(6)                VALUE "OVRFLO".
01      TMP                     PIC X(10).
01      RETURNED_NUM_LINES      PIC XX.
01      NUM_LINES_IN_SCROLL     PIC 99          COMP.
PROCEDURE DIVISION.
0.
* Put up register form.
* Check for current session totals overflow. If so, output 'OVRFLO'
* Put out summary of this session into indexed(4) fields.
*-
        CALL "FDV$CDISP" USING  BY DESCRIPTOR   FORM-REGIST.
        CALL "SRVCHK".
        MOVE 1 TO FIELD_INDEX.
        CALL "FDV$PUT" USING    BY DESCRIPTOR   ACCT_BALANCE
                                BY DESCRIPTOR   N-REGIST-SUMARY
                                BY REFERENCE    FIELD_INDEX.
        MOVE 2 TO FIELD_INDEX.
        IF TOTAL_DEPOSIT IS NOT GREATER THAN MAX_DEPOSIT
                THEN    CALL "FDV$PUT" USING    BY DESCRIPTOR   TOTAL_DEPOSIT
                                                BY DESCRIPTOR   N-REGIST-SUMARY
                                                BY REFERENCE    FIELD_INDEX
                ELSE    CALL "FDV$PUT" USING    BY DESCRIPTOR   OVERFLOW_MSG
                                                BY DESCRIPTOR   N-REGIST-SUMARY
                                                BY REFERENCE    FIELD_INDEX.
        MOVE 3 TO FIELD_INDEX.
        IF TOTAL_PAYMENTS IS LESS THAN MAX_PAYMENT
                THEN    CALL "FDV$PUT" USING    BY DESCRIPTOR   TOTAL_PAYMENTS
                                                BY DESCRIPTOR   N-REGIST-SUMARY
                                                BY REFERENCE    FIELD_INDEX
                ELSE    CALL "FDV$PUT" USING    BY DESCRIPTOR   OVERFLOW_MSG
                                                BY DESCRIPTOR   N-REGIST-SUMARY
                                                BY REFERENCE    FIELD_INDEX.
        MOVE 4 TO FIELD_INDEX.
        CALL "FDV$PUT"  USING   BY DESCRIPTOR   CURRENT_BALANCE
                                BY DESCRIPTOR   N-REGIST-SUMARY
                                BY REFERENCE    FIELD_INDEX.
*+
* Get number of lines in scroll area from form named data (item 1).
*-
        MOVE 1 TO FIELD_INDEX.
        CALL "FDV$RETDI" USING  BY REFERENCE    FIELD_INDEX
                                BY DESCRIPTOR   RETURNED_NUM_LINES.
        CALL "SRVCHK".
        IF RETURNED_NUM_LINES(2:1) = SPACE
```

```
*+
* Put lines from check register array into scrolled area.
* The window is initially from item 1 up to item
* min(NUM_LINES_IN_SCROLL,LAST_REGISTER_NUM), that is, up to the size of the scrolled
* area or the size of the register, whichever is less. Assume there
* is at least one line (the initial deposit).
*-
        MOVE FDV$K_FT_SFW TO TERMINATOR.
        PERFORM WITH TEST AFTER VARYING CUR_LINE FROM 1 BY 1
                UNTIL CUR_LINE NOT LESS NUM_LINES_IN_SCROLL OR CUR_LINE NOT LESS LAST_REGISTER_NUM
                IF CUR_LINE NOT = 1 THEN CALL "FDV$PFT" USING   BY REFERENCE    TERMINATOR
                                                                BY DESCRIPTOR   N-CHECK-NUMBER
                END-IF
                CALL "FDV$PUTSC" USING  BY DESCRIPTOR   N-CHECK-NUMBER
                                        BY DESCRIPTOR   REGISTER_ITEM(CUR_LINE)
        END-PERFORM.
* Set the MIN and MAX window.
        MOVE 1 TO MIN_WINDOW.
        MOVE CUR_LINE TO MAX_WINDOW.
*+
* Get input from fake field of scrolled line and do what it says:
*       Kpd . or RETURN/ENTER   => return to menu
*       UPARROW or TAB          => scroll forward
*       DOWNARROW or BACKSPACE  =  scroll backward
*       all others                 => ignore
* Note that there is no form function key UAR so this routine
* handles all terminators itself (by ignoring illegal ones).
*-
        MOVE FDV$K_FT_SBK TO TERMINATOR.
        PERFORM UNTIL TERMINATOR = FDV$K_FT_NTR OR TERMINATOR = FDV$K_KP_PER
                CALL "FDV$GET" USING    BY DESCRIPTOR   TMP
                                        BY REFERENCE    TERMINATOR
                                        BY DESCRIPTOR   N-REGIST-FAKE
                EVALUATE        TERMINATOR
                        WHEN    FDV$K_FT_SBK
                        WHEN    FDV$K_FT_SPR    CALL "SCRBAK"
                        WHEN    FDV$K_FT_SFW
                        WHEN    FDV$K_FT_SNX    CALL "SCRFWD"
                END-EVALUATE
        END-PERFORM.
        EXIT PROGRAM.
```

```
IDENTIFICATION DIVISION.
PROGRAM-ID.     SCRFWD.
*+
*       CUR_LINE is the line in the register that the cursor is on.
*       MIN_WINDOW and MAX_WINDOW delimit the part of the register
*       currently displayed in the scrolled area.
*-
DATA DIVISION.
WORKING-STORAGE SECTION.
01      END_OF_REG_MSG          PIC X(21)               VALUE "Last line of register".
PROCEDURE DIVISION.
0.
*+
* If cursor is at the end of the register, report, and return
* If cursor not at the last line of a window, just move down
* If cursor is  at the last line of a window,
*       move window forward one line,
*       write the new last line to the last line of the scrolled area
* Move current line pointer forward
*-
        IF CUR_LINE = LAST_REGISTER_NUM
                THEN    CALL "FDV$PUT_" USING   BY DESCRIPTOR   END_OF_REG_MSG
                ELSE    MOVE FDV$K_FT_SFW TO TERMINATOR
                        IF CUR_LINE NOT = MAX_WINDOW
                                THEN    CALL "FDV$PFT" USING    BY REFERENCE    TERMINATOR
                                                                BY DESCRIPTOR   N-CHECK-NUMBER
                                ELSE    ADD 1 TO MIN_WINDOW, MAX_WINDOW
                                        CALL "FDV$PFT" USING    BY REFERENCE    TERMINATOR
                                                                BY DESCRIPTOR   N-CHECK-NUMBER
                                                                BY DESCRIPTOR   REGISTER_ITEM(MAX_WINDOW)
                                END-IF
                        ADD 1 TO CUR_LINE
                END-IF.
        EXIT PROGRAM.
END PROGRAM SCRFWD.
```

```
IDENTIFICATION DIVISION.
PROGRAM-ID.     SCRBAK.
*+
*       CUR_LINE is the line in the register that the cursor is on.
*       MIN_WINDOW and MAX_WINDOW delimit the part of the register
*       currently displayed in the scrolled area
*-
DATA DIVISION.
WORKING-STORAGE SECTION.
01      START_OF_REG_MSG        PIC X(22)               VALUE "First line of register".
PROCEDURE DIVISION.
0.
```

```
*+
* If the cursor is at the beginning of the register, report, and return
* If cursor not at first line of the window, just move up
* If cursor is  at first line of the window,
*       move window back one line,
*       write the new first line to the first line of the scrolled area
* Move current line pointer back
*-
        IF CUR_LINE = 1
                THEN    CALL "FDV$PUTL" USING   BY DESCRIPTOR   START_OF_REG_MSG
                ELSE    MOVE FDV$K_FT_SBK TO TERMINATOR
                        IF CUR_LINE NOT = MIN_WINDOW
                                THEN    CALL "FDV$PFT" USING    BY REFERENCE    TERMINATOR
                                                                BY DESCRIPTOR   N-CHECK-NUMBER
                                ELSE    SUBTRACT 1 FROM MIN_WINDOW, MAX_WINDOW
                                        CALL "FDV$PFT" USING    BY REFERENCE    TERMINATOR
                                                                BY DESCRIPTOR   N-CHECK-NUMBER
                                                                BY DESCRIPTOR   REGISTER_ITEM(MIN_WINDOW)
                                END-IF
                        SUBTRACT 1 FROM CUR_LINE
                END-IF.
        EXIT PROGRAM.
END PROGRAM SCRBAK.
END PROGRAM VUEREG.
END PROGRAM MENU.



IDENTIFICATION DIVISION.
PROGRAM-ID.     SRVCHK IS COMMON.
*+
* Check FMS status by looking at the status recording variables.
*-
DATA DIVISION.
WORKING-STORAGE SECTION.
01      TMP                     PIC 9(9).
PROCEDURE DIVISION.
0.
        IF FMS_STATUS LESS THAN ZERO
        THEN
*+
* There is an error returned in the status variables. Detach the
* terminal to clean up, then print the errors, and stop.
*
                CALL "FDV$DTERM"  USING BY DESCRIPTOR   TERM_CONTROL_AREA
                DISPLAY SPACE
                DISPLAY "FDV ERROR"
                MOVE FMS_STATUS TO TMP
                DISPLAY "FMS STATUS: ",TMP
                MOVE RMS_STATUS TO TMP
                DISPLAY "RMS STATUS: ",TMP
                STOP RUN.
        EXIT PROGRAM.
END PROGRAM SRVCHK.
```

```
IDENTIFICATION DIVISION.
PROGRAM-ID.     GETSTAT IS COMMON.
*+
* Check FMS status by calling FDV$STAT.
* If not success (>0), print and stop
*-
*
DATA DIVISION.
WORKING-STORAGE SECTION.
PROCEDURE DIVISION.
0.
        CALL "SRVCHK".
        EXIT PROGRAM.
END PROGRAM GETSTAT.
```

```
IDENTIFICATION DIVISION.
PROGRAM-ID.     FMTCHK IS COMMON.
*+
*       Format account data onto check form in the check workspace.
*-
DATA DIVISION.
WORKING-STORAGE SECTION.
01      NAME_CONDENSED          PIC X(39).
01      FIRST_LEN               PIC 9(4)        COMP.
01      CSZ_CONDENSED           PIC X(30).
01      CITY_LEN                PIC 9(4)        COMP.
01      UNUSED_STRING           PIC X(80).
PROCEDURE DIVISION.
0.
        CALL "FDV$SWKSP" USING  BY DESCRIPTOR   CHECK_WORKSPACE.
        CALL "FDV$LOAD" USING   BY DESCRIPTOR   FORM-CHECK.
*+
* Need to trim trailing blanks - use the VMS RTL routine to find out how
* long the trimmed string is, then do explicit moves.
* Put only middle initial in, not full middle name.
*-
        CALL "STR$TRIM" USING   BY DESCRIPTOR   UNUSED_STRING
                                BY DESCRIPTOR   FIRST_NAME
                                BY REFERENCE    FIRST_LEN.

        STRING  FIRST_NAME(1:FIRST_LEN) " "
                MIDDLE_NAME(1:1) ". "
                LAST_NAME DELIMITED BY SIZE INTO NAME_CONDENSED.

        CALL "FDV$PUT" USING    BY DESCRIPTOR   NAME_CONDENSED
                                BY DESCRIPTOR   N-CHECK-NAME.
        CALL "FDV$PUT" USING    BY DESCRIPTOR   ACCT_STREET
                                BY DESCRIPTOR   N-CHECK-STREET.
```

```
        CALL "STR$TRIM" USING   BY DESCRIPTOR   UNUSED_STRING
                                BY DESCRIPTOR   CITY
                                BY REFERENCE    CITY_LEN.
        STRING  CITY(1:CITY_LEN) ", "
                STATE " "
                ZIP             DELIMITED BY SIZE INTO CSZ_CONDENSED.

        CALL "FDV$PUT" USING    BY DESCRIPTOR   CSZ_CONDENSED
                                BY DESCRIPTOR   N-CHECK-CSZ.
        CALL "FDV$PUT" USING    BY DESCRIPTOR   ACCT_HOME_PHONE
                                BY DESCRIPTOR   N-CHECK-HOMEPH.
        CALL "FDV$PUT" USING    BY DESCRIPTOR   ACCT_NUMBER
                                BY DESCRIPTOR   N-CHECK-ACCTNO.
        CALL "FDV$SWKSP" USING  BY DESCRIPTOR   WORKSPACE.
        EXIT PROGRAM.
END PROGRAM FMTCHK.
END PROGRAM SAMP.



IDENTIFICATION DIVISION.
PROGRAM-ID.     VALID1  INITIAL.
*****************************************************************************
*   Field completion UAR for field validation of any one character field. The *
*   UAR associated data has in it the legal characters allowed,               *
*   except that blank is not allowed unless it appears before                 *
*   the first trailing blank.  For example an assoc. value string             *
*   'aqr' implies that only the letters a, q, and r are allowed.              *
*   A string ' aqr' means that blank is acceptable in addition to a, q and r. *
*   Note that this routine is case sensitive                                  *
*   (that is, it checks for correct case).  You can get around                *
*   case sensitivity by using the force upper case field attribute            *
*   and putting only capitals into the UAR associated value                   *
*   strings.                                                                  *
*                                                                             *
*   This routine can be used with any form and field since                    *
*   it determines the context for itself.                                     *
*****************************************************************************
DATA DIVISION.
WORKING-STORAGE SECTION.
        COPY "FDVDEF".
        COPY "SMPCOBUAR".
*
*   Declarations specfic to this UAR.
*
01      FIELD_VALUE             PIC X(1).
01      COUNTER                 PIC 9(2)        COMP    VALUE 0.
01      ILLEGAL_VALUE_MSG       PIC X(13)               VALUE "Illegal value".
```

```
PROCEDURE DIVISION GIVING RETURN_STATUS.
0.
*+
* Retrieve context: ignore all but UAR_DATA, and
* only the initial, non-blank characters of it.
* Retrieve field name and index.
* Retrieve field value.
*-
        CALL "FDV$RETCX"        USING   BY REFERENCE    ADDRESS_TCA,
                                        BY REFERENCE    ADDRESS_WKSP,
                                        BY DESCRIPTOR   FORM_NAME,
                                        BY DESCRIPTOR   UAR_DATA,
                                        BY REFERENCE    CURSOR_POSITION,
                                        BY REFERENCE    TERMINATOR,
                                        BY REFERENCE    INSOVR_STATUS,
                                        BY REFERENCE    HELP_STRIKES.

        CALL "FDV$RETFN"        USING   BY DESCRIPTOR   FIELD_NAME,
                                        BY REFERENCE    FIELD_INDEX.

        CALL "FDV$RET"          USING   BY DESCRIPTOR   FIELD_VALUE,
                                        BY DESCRIPTOR   FIELD_NAME,
                                        BY REFERENCE    FIELD_INDEX.
*+
* To be valid, FIELD_VALUE must occur in the string UAR_DATA.
* This INSPECT statement sets COUNTER to the number of characters preceeding
* FIELD_VALUE; thus, COUNTER will be 0 if the first character in UAR_DATA
* matched and UAR_DATA_LENGTH if the character is not found.
*-
        INSPECT UAR_DATA TALLYING COUNTER FOR CHARACTERS BEFORE INITIAL FIELD_VALUE.
        IF COUNTER IS LESS THAN UAR_DATA_MAX_LENGTH
        THEN
                MOVE FDV$K_UVAL_SUC TO RETURN_STATUS
        ELSE
                CALL "FDV$PUTL" USING   BY DESCRIPTOR   ILLEGAL_VALUE_MSG
                MOVE FDV$K_UVAL_FAIL TO RETURN_STATUS
        END-IF.

        EXIT PROGRAM.
END PROGRAM VALID1.
```

```
IDENTIFICATION DIVISION.
PROGRAM-ID.     TAKE15  INITIAL.
*****************************************************************
*   Function Key User Action Routine for the MENU form of SAMP.  *
*   Convert keypad 1-5 into field values 1-5.                    *
*   Convert keypad period into field value 1.                    *
*   Reject all other function keys with error message.           *
*****************************************************************
DATA DIVISION.
WORKING-STORAGE SECTION.
        COPY "FDVDEF".
        COPY "SAMPCOB".
        COPY "SMPCOBUAR".
*
*   Declarations specfic to this UAR.
*
01      FIELD_VALUE             PIC X(1)                VALUE SPACE.
01      ILLEGAL_FUNC_KEY_MSG    PIC X(20)               VALUE "Illegal function key".
PROCEDURE DIVISION      GIVING  RETURN_STATUS.
0.
*+
* Retrieve context: ignore all but TERMINATOR
*-
        CALL "FDV$RETCX"        USING   BY REFERENCE    ADDRESS_TCA,
                                        BY REFERENCE    ADDRESS_WKSP,
                                        BY DESCRIPTOR   FORM_NAME,
                                        BY DESCRIPTOR   UAR_DATA,
                                        BY REFERENCE    CURSOR_POSITION,
                                        BY REFERENCE    TERMINATOR,
                                        BY REFERENCE    INSOVR_STATUS,
                                        BY REFERENCE    HELP_STRIKES.
*+
* Do the conversion, displaying the value converted if found.
* Reject if not one of the expected terminators.
*
        EVALUATE TERMINATOR
                WHEN    FDV$K_KP_1      MOVE "1" TO FIELD_VALUE
                WHEN    FDV$K_KP_2      MOVE "2" TO FIELD_VALUE
                WHEN    FDV$K_KP_3      MOVE "3" TO FIELD_VALUE
                WHEN    FDV$K_KP_4      MOVE "4" TO FIELD_VALUE
                WHEN    FDV$K_KP_5      MOVE "5" TO FIELD_VALUE
                WHEN    FDV$K_KP_PER    MOVE "1" TO FIELD_VALUE
        END-EVALUATE.
        IF FIELD_VALUE = SPACE THEN
           CALL "FDV$PUTL"      USING   BY DESCRIPTOR ILLEGAL_FUNC_KEY_MSG
           CALL "FDV$SIGOP"
*          Just ignore it now.
           MOVE FDV$K_UKEY_SUC TO RETURN_STATUS
        ELSE
           CALL "FDV$PUT"       USING   BY DESCRIPTOR FIELD_VALUE
                                        BY DESCRIPTOR N-MENU-OPTION
*          Treat as if it is RETURN.
           MOVE FDV$K_UKEY_NTR TO RETURN_STATUS
        END-IF.
        EXIT PROGRAM.
```

```
IDENTIFICATION DIVISION.
PROGRAM-ID.     PASSKY  INITIAL.
*****************************************************************************
*   General function key UAR to pass only those from the (small) list       *
*   in the uar associated value string and reject all others.               *
*   The list is of the form: n 'oneblank> n <oneblank> ... n <manyblanks'   *
*   For example the string '110 112' would accept keypad period and         *
*   keypad zero but no other function keys.                                 *
*****************************************************************************
DATA DIVISION.
WORKING-STORAGE SECTION.
        COPY "FDVDEF".
        COPY "SMPCOBUAR".
*
*   Declarations specfic to this UAR.
*
01      ITEM            PIC 9(4)                VALUE IS ZERO.
01      ITEM_LENGTH     PIC 9(4)        COMP    VALUE IS ZERO.
01      NON_BLANK_PTR   PIC 9(4)        COMP    VALUE 1.
01      DONE            PIC S9(9)       COMP.
PROCEDURE DIVISION      GIVING  RETURN_STATUS.
0.
*+
* Retrieve context: ignore all but TERMINATOR and UAR_DATA.
*-
        CALL "FDV$RETCX"        USING   BY REFERENCE    ADDRESS_TCA,
                                        BY REFERENCE    ADDRESS_WKSP,
                                        BY DESCRIPTOR   FORM_NAME,
                                        BY DESCRIPTOR   UAR_DATA,
                                        BY REFERENCE    CURSOR_POSITION,
                                        BY REFERENCE    TERMINATOR,
                                        BY REFERENCE    INSOVR_STATUS,
                                        BY REFERENCE    HELP_STRIKES.
*+
* Break up the list into numbers.  Check each against the actual
* terminator. If terminator found in list, return success.
*-
        SET DONE TO FAILURE.
        MOVE FDV$K_UKEY_ERR TO RETURN_STATUS.

        PERFORM WITH TEST AFTER UNTIL DONE IS SUCCESS OR ITEM = ZERO
                UNSTRING UAR_DATA DELIMITED BY SPACE INTO ITEM
                        WITH POINTER NON_BLANK_PTR
                IF NON_BLANK_PTR IS GREATER THAN UAR_DATA_MAX_LENGTH
                    SET DONE TO SUCCESS
                END-IF
                IF TERMINATOR = ITEM THEN
                    MOVE FDV$K_UKEY_TRM TO RETURN_STATUS
                    SET DONE TO SUCCESS
                END-IF
        END-PERFORM.

        EXIT PROGRAM
```

```
IDENTIFICATION DIVISION.
PROGRAM-ID.     CHKCHK  INITIAL.
*****************************************************************
*   Field completion UAR for SAMP CHECK form.  Makes sure that the      *
*   check amount is less than or equal to the current balance.  If not, *
*   complain and change video attributes on balance field so the        *
*   potential bouncer can see what there is to work with.               *
*****************************************************************
DATA DIVISION.
WORKING-STORAGE SECTION.
        COPY "FDVDEF".
        COPY "SAMPCOB".
        COPY "SMPCOBUAR".
*
*   Declarations specfic to this UAR.
*
01      CURRENT_BALANCE         PIC 9(6).
01      AMOUNT                  PIC 9(6).
01      BLINKBOLD               PIC 9(6)        COMP.
01      OVERDRAFT_MSG           PIC X(53)               VALUE "Your balance doesn't cover that much, reenter amount.".
PROCEDURE DIVISION      GIVING RETURN_STATUS.
0.
        CALL "FDV$RET"  USING   BY DESCRIPTOR   CURRENT_BALANCE,
                                BY DESCRIPTOR   N-CHECK-BALANC.
        CALL "FDV$RET"  USING   BY DESCRIPTOR   AMOUNT,
                                BY DESCRIPTOR   N-CHECK-AMTPAY.
        INSPECT AMOUNT REPLACING ALL SPACES BY ZERO.

        IF CURRENT_BALANCE IS NOT LESS THAN AMOUNT THEN
                MOVE -1 TO BLINKBOLD
                CALL "FDV$AFVA" USING   BY REFERENCE    BLINKBOLD,
                                        BY DESCRIPTOR   N-CHECK-BALANC
                MOVE FDV$K_UVAL_SUC TO RETURN_STATUS
        ELSE
            MOVE 3 TO BLINKBOLD
            CALL "FDV$AFVA"     USING   BY REFERENCE    BLINKBOLD,
                                        BY DESCRIPTOR   N-CHECK-BALANC
            CALL "FDV$PUTL"     USING   BY DESCRIPTOR   OVERDRAFT_MSG
            MOVE FDV$K_UVAL_FAIL TO RETURN_STATUS
        END-IF.
        EXIT PROGRAM.
END PROGRAM CHKCHK.
```

```
IDENTIFICATION DIVISION.
PROGRAM-ID.     RANGE   INITIAL.
*****************************************************************************
*   General purpose field completion UAR to check the range of any numeric  *
*   item.  The associated UAR data must have one of the four forms:         *
*                                                                           *
*         L,U<space>{message}                                               *
*         ,U<space>{message}                                                *
*         L,<space>{message}                                                *
*         ,<space>{message}                                                 *
*                                                                           *
*   where L is lower bound, U is upper bound, and {message} is an           *
*   optional error message in case the field value is out of bounds.        *
*   If one of the bounds isn't given, it isn't checked for.  If neither     *
*   bound is given, nothing is checked, everything succeeds.  If the        *
*   UAR value doesn't have a comma, COBOL issues a run-time error.          *
*   The form designer has to go                                             *
*   back and do it right. If no {message} is given, a simple                *
*   "out of range U:L" message is given to the hapless operator.            *
*                                                                           *
*   This UAR can work with any form and numeric field since it gets         *
*   context itself.  Care must be taken with fields using field marker      *
*   periods since those periods are not returned to the program.            *
*****************************************************************************
DATA DIVISION.
WORKING-STORAGE SECTION.
        COPY "FDVDEF".
        COPY "SMPCOBUAR".
*
*   Declarations specfic to this UAR.
*
01      FLD_NUMBER              PIC X(132).
01      FLD_LENGTH              PIC 9(9)        COMP.
01      JUSTIFIED_NUMBER        PIC S9(18).
01      MAX_NUMERIC_CHARS       PIC 9(2)        COMP    VALUE 19.
01      LOWER                   PIC S9(18).
01      UPPER                   PIC S9(18).
01      LOWER_COUNT             PIC 9(2).
01      UPPER_COUNT             PIC 9(2).
01      BEGIN_MSG_PTR           PIC 9(9)        COMP    VALUE 1.
01      IN_RANGE                PIC S9(9)       COMP.
01      CANNED_MSG              PIC X(45)               VALUE "Field value out of bounds.  Must be in range ".
01      CANNED_PTR              PIC 9(9)        COMP    VALUE 1.
01      BAD_MSG                 PIC X(80).
01      MSG_STRING              PIC X(80).
01      MSG_COUNT               PIC 9(3)        COMP.
PROCEDURE DIVISION      GIVING RETURN_STATUS.
0.
*-
* Get context which yields associated data value (ignore other stuff).
* Get current field name and index.
* Get field value.
```

```
        CALL "FDV$RETCX"        USING   BY REFERENCE    ADDRESS_TCA,
                                        BY REFERENCE    ADDRESS_WKSP,
                                        BY DESCRIPTOR   FORM_NAME,
                                        BY DESCRIPTOR   UAR_DATA,
                                        BY REFERENCE    CURSOR_POSITION,
                                        BY REFERENCE    TERMINATOR,
                                        BY REFERENCE    INSOVR_STATUS,
                                        BY REFERENCE    HELP_STRIKES.

        CALL "FDV$RETFN"        USING   BY DESCRIPTOR   FIELD_NAME,
                                        BY REFERENCE    FIELD_INDEX.

        CALL "FDV$RET"          USING   BY DESCRIPTOR   FLD_NUMBER,
                                        BY DESCRIPTOR   FIELD_NAME,
                                        BY REFERENCE    FIELD_INDEX.

        CALL "FDV$RETLE"        USING   BY REFERENCE    FLD_LENGTH,
                                        BY DESCRIPTOR   FIELD_NAME.

*+
* COBOL numbers must be no longer than 18 digits plus sign.
* Also, all numbers are assumed to be integers.
*-
        IF FLD_LENGTH IS GREATER THAN MAX_NUMERIC_CHARS THEN
           SET RETURN_STATUS TO FAILURE
        ELSE
           INSPECT FLD_NUMBER(1:FLD_LENGTH) REPLACING ALL SPACES BY ZERO
           MOVE FLD_NUMBER(1:FLD_LENGTH) TO JUSTIFIED_NUMBER
*+
* Find comma and blank delimiters.
*-
           UNSTRING UAR_DATA DELIMITED BY "," OR SPACE
                   INTO LOWER COUNT IN LOWER_COUNT
                        UPPER COUNT IN UPPER_COUNT
                   WITH POINTER BEGIN_MSG_PTR

           SET IN_RANGE TO SUCCESS
*+
* Check for lower bound.
*-
           IF LOWER_COUNT IS NOT = ZERO THEN
              IF JUSTIFIED_NUMBER IS LESS THAN LOWER SET IN_RANGE TO FAILURE
           END-IF
*+
* Check for upper bound
*-
           IF UPPER_COUNT IS NOT = ZERO THEN
              IF JUSTIFIED_NUMBER IS GREATER THAN UPPER SET IN_RANGE TO FAILURE
           END-IF
           END-IF
```

```
*+
* Passed both tests successfully, return success for UAR value
*-
           IF IN_RANGE IS SUCCESS THEN
              MOVE FDV$K_UVAL_SUC TO RETURN_STATUS
           ELSE
*+
* Error in one of the bounds.
* Give error message: either from the UARVAL or make one up.
*-
              UNSTRING UAR_DATA DELIMITED BY "  "
                 INTO MSG_STRING COUNT IN MSG_COUNT
                 WITH POINTER BEGIN_MSG_PTR

              IF MSG_COUNT IS GREATER THAN ZERO THEN
                 CALL "FDV$PUTL"        USING   BY DESCRIPTOR   MSG_STRING(1:MSG_COUNT)
              ELSE
*+
* BEGIN_MSG_PTR is too long since (a) the message begins 2 characters past the end
* of the lower--upper-bound string, and (b) the above UNSTRING results in another
* incrementing of 2
*-
                 SUBTRACT 4 FROM BEGIN_MSG_PTR
                 STRING CANNED_MSG UAR_DATA(1:BEGIN_MSG_PTR) "."
                   DELIMITED BY SIZE INTO
                   BAD_MSG WITH POINTER CANNED_PTR
                 CALL "FDV$PUTL"        USING   BY DESCRIPTOR   BAD_MSG(1:CANNED_PTR)
              END-IF

              CALL "FDV$SIGOP"
              MOVE FDV$K_UVAL_FAIL TO RETURN_STATUS
           END-IF
       END-IF.
       EXIT PROGRAM.
END PROGRAM RANGE.
```

```
*****************************************************************************
* FDVDEF.LIB  --  This is the copy file for FMS applications in COBOL. *
*****************************************************************************
*************************
*************************
01      FDV$K_FT_NTR            PIC S9(9)  COMP GLOBAL  VALUE IS  0.
01      FDV$K_FT_NXT            PIC S9(9)  COMP GLOBAL  VALUE IS  1.
01      FDV$K_FT_PRV            PIC S9(9)  COMP GLOBAL  VALUE IS  2.
01      FDV$K_FT_ATB            PIC S9(9)  COMP GLOBAL  VALUE IS  3.
01      FDV$K_FT_XBK            PIC S9(9)  COMP GLOBAL  VALUE IS  4.
01      FDV$K_FT_XFW            PIC S9(9)  COMP GLOBAL  VALUE IS  5.
01      FDV$K_FT_SNX            PIC S9(9)  COMP GLOBAL  VALUE IS  6.
01      FDV$K_FT_SPR            PIC S9(9)  COMP GLOBAL  VALUE IS  7.
01      FDV$K_FT_SFW            PIC S9(9)  COMP GLOBAL  VALUE IS  8.
01      FDV$K_FT_SBK            PIC S9(9)  COMP GLOBAL  VALUE IS  9.
01      FDV$K_FT_ILG_NXT        PIC S9(9)  COMP GLOBAL  VALUE IS 11.
01      FDV$K_FT_ILG_PRV        PIC S9(9)  COMP GLOBAL  VALUE IS 12.
01      FDV$K_FT_ILG_ATB        PIC S9(9)  COMP GLOBAL  VALUE IS 13.
01      FDV$K_FT_ILG_XBK        PIC S9(9)  COMP GLOBAL  VALUE IS 14.
01      FDV$K_FT_ILG_XFW        PIC S9(9)  COMP GLOBAL  VALUE IS 15.
01      FDV$K_FT_ILG_SFW        PIC S9(9)  COMP GLOBAL  VALUE IS 16.
01      FDV$K_FT_ILG_SBK        PIC S9(9)  COMP GLOBAL  VALUE IS 17.
*****************************************************
* Function Key terminators returned from GETs and WAIT *
* Also used as FDV Keycodes for use with DFKBD.       *
*****************************************************
01      FDV$K_AR_UP             PIC S9(9)  COMP GLOBAL  VALUE IS  99.
01      FDV$K_AR_DOWN           PIC S9(9)  COMP GLOBAL  VALUE IS 100.
01      FDV$K_AR_LEFT           PIC S9(9)  COMP GLOBAL  VALUE IS 101.
01      FDV$K_AR_RIGHT          PIC S9(9)  COMP GLOBAL  VALUE IS 102.
01      FDV$K_PF_1              PIC S9(9)  COMP GLOBAL  VALUE IS 103.
01      FDV$K_PF_2              PIC S9(9)  COMP GLOBAL  VALUE IS 104.
01      FDV$K_PF_3              PIC S9(9)  COMP GLOBAL  VALUE IS 105.
01      FDV$K_PF_4              PIC S9(9)  COMP GLOBAL  VALUE IS 106.
01      FDV$K_KP_NTR            PIC S9(9)  COMP GLOBAL  VALUE IS 107.
01      FDV$K_KP_COM            PIC S9(9)  COMP GLOBAL  VALUE IS 108.
01      FDV$K_KP_HYP            PIC S9(9)  COMP GLOBAL  VALUE IS 109.
01      FDV$K_KP_PER            PIC S9(9)  COMP GLOBAL  VALUE IS 110.
01      FDV$K_KP_0              PIC S9(9)  COMP GLOBAL  VALUE IS 112.
01      FDV$K_KP_1              PIC S9(9)  COMP GLOBAL  VALUE IS 113.
01      FDV$K_KP_2              PIC S9(9)  COMP GLOBAL  VALUE IS 114.
01      FDV$K_KP_3              PIC S9(9)  COMP GLOBAL  VALUE IS 115.
01      FDV$K_KP_4              PIC S9(9)  COMP GLOBAL  VALUE IS 116.
01      FDV$K_KP_5              PIC S9(9)  COMP GLOBAL  VALUE IS 117.
01      FDV$K_KP_6              PIC S9(9)  COMP GLOBAL  VALUE IS 118.
01      FDV$K_KP_7              PIC S9(9)  COMP GLOBAL  VALUE IS 119.
01      FDV$K_KP_8              PIC S9(9)  COMP GLOBAL  VALUE IS 120.
01      FDV$K_KP_9              PIC S9(9)  COMP GLOBAL  VALUE IS 121.
01      FDV$K_GAR_UP            PIC S9(9)  COMP GLOBAL  VALUE IS 227.
01      FDV$K_GAR_DOWN          PIC S9(9)  COMP GLOBAL  VALUE IS 228.
01      FDV$K_GAR_RIGHT         PIC S9(9)  COMP GLOBAL  VALUE IS 229.
01      FDV$K_GAR_LEFT          PIC S9(9)  COMP GLOBAL  VALUE IS 230.
01      FDV$K_GPF_1             PIC S9(9)  COMP GLOBAL  VALUE IS 231.
```

```
01      FDV$K_GPF_2             PIC S9(9)  COMP GLOBAL  VALUE IS  232.
01      FDV$K_GPF_3             PIC S9(9)  COMP GLOBAL  VALUE IS  233.
01      FDV$K_GPF_4             PIC S9(9)  COMP GLOBAL  VALUE IS  234.
01      FDV$K_GKP_NTR           PIC S9(9)  COMP GLOBAL  VALUE IS  235.
01      FDV$K_GKP_COM           PIC S9(9)  COMP GLOBAL  VALUE IS  236.
01      FDV$K_GKP_HYP           PIC S9(9)  COMP GLOBAL  VALUE IS  237.
01      FDV$K_GKP_PER           PIC S9(9)  COMP GLOBAL  VALUE IS  238.
01      FDV$K_GKP_0             PIC S9(9)  COMP GLOBAL  VALUE IS  240.
01      FDV$K_GKP_1             PIC S9(9)  COMP GLOBAL  VALUE IS  241.
01      FDV$K_GKP_2             PIC S9(9)  COMP GLOBAL  VALUE IS  242.
01      FDV$K_GKP_3             PIC S9(9)  COMP GLOBAL  VALUE IS  243.
01      FDV$K_GKP_4             PIC S9(9)  COMP GLOBAL  VALUE IS  244.
01      FDV$K_GKP_5             PIC S9(9)  COMP GLOBAL  VALUE IS  245.
01      FDV$K_GKP_6             PIC S9(9)  COMP GLOBAL  VALUE IS  246.
01      FDV$K_GKP_7             PIC S9(9)  COMP GLOBAL  VALUE IS  247.
01      FDV$K_GKP_8             PIC S9(9)  COMP GLOBAL  VALUE IS  248.
01      FDV$K_GKP_9             PIC S9(9)  COMP GLOBAL  VALUE IS  249.
*****************************************
* FDV Keyfunctions. For use in DFKBD call. *
*****************************************
01      FDV$K_KF_GOLD   PIC S9(9)  COMP GLOBAL  VALUE IS   1.
01      FDV$K_KF_RESET  PIC S9(9)  COMP GLOBAL  VALUE IS   2.
01      FDV$K_KF_CRSLF  PIC S9(9)  COMP GLOBAL  VALUE IS   3.
01      FDV$K_KF_CRSRT  PIC S9(9)  COMP GLOBAL  VALUE IS   4.
01      FDV$K_KF_DLCHR  PIC S9(9)  COMP GLOBAL  VALUE IS   5.
01      FDV$K_KF_DLFLD  PIC S9(9)  COMP GLOBAL  VALUE IS   6.
01      FDV$K_KF_INS    PIC S9(9)  COMP GLOBAL  VALUE IS   7.
01      FDV$K_KF_OVR    PIC S9(9)  COMP GLOBAL  VALUE IS   8.
01      FDV$K_KF_RFRSH  PIC S9(9)  COMP GLOBAL  VALUE IS   9.
01      FDV$K_KF_HELP   PIC S9(9)  COMP GLOBAL  VALUE IS  10.
01      FDV$K_KF_NXT    PIC S9(9)  COMP GLOBAL  VALUE IS  11.
01      FDV$K_KF_PRV    PIC S9(9)  COMP GLOBAL  VALUE IS  12.
01      FDV$K_KF_NTR    PIC S9(9)  COMP GLOBAL  VALUE IS  13.
01      FDV$K_KF_SBK    PIC S9(9)  COMP GLOBAL  VALUE IS  14.
01      FDV$K_KF_SFW    PIC S9(9)  COMP GLOBAL  VALUE IS  15.
01      FDV$K_KF_XBK    PIC S9(9)  COMP GLOBAL  VALUE IS  16.
01      FDV$K_KF_XFW    PIC S9(9)  COMP GLOBAL  VALUE IS  17.
01      FDV$K_KF_NONE   PIC S9(9)  COMP GLOBAL  VALUE IS   0.
01      FDV$K_KF_DFLT   PIC S9(9)  COMP GLOBAL  VALUE IS  -1.
*************************************************************
* UAR return codes. These codes are returned by UAR to FDV. *
*************************************************************
* Field completion return codes *
*********************************
01      FDV$K_UVAL_SUC  PIC S9(9)  COMP GLOBAL  VALUE IS  1000.
01      FDV$K_UVAL_FAIL PIC S9(9)  COMP GLOBAL  VALUE IS  1001.
01      FDV$K_UVAL_END  PIC S9(9)  COMP GLOBAL  VALUE IS  1002.
************************
* Help UAR return codes *
************************
01      FDV$K_UHELP_NO  PIC S9(9)  COMP GLOBAL  VALUE IS  2000.
01      FDV$K_UHELPED   PIC S9(9)  COMP GLOBAL  VALUE IS  2001.
```

```
*********************************
* Function Key UAR return codes *
*********************************
01      FDV$K_UKEY_ERR  PIC S9(9)  COMP GLOBAL  VALUE IS  3000.
01      FDV$K_UKEY_TRM  PIC S9(9)  COMP GLOBAL  VALUE IS  3001.
01      FDV$K_UKEY_NXT  PIC S9(9)  COMP GLOBAL  VALUE IS  3002.
01      FDV$K_UKEY_NTR  PIC S9(9)  COMP GLOBAL  VALUE IS  3003.
01      FDV$K_UKEY_SUC  PIC S9(9)  COMP GLOBAL  VALUE IS  3004.
*****************************************************************************
* FDV status codes returned when FDV$... routines are called as functions.  *
* These codes are VMS status codes and can be signalled.  They correspond   *
* one-to-one with the FMS status codes retrievable from FDV$STAT.           *
*****************************************************************************
01      FDV$_SUC        PIC S9(9)  COMP GLOBAL  VALUE IS 2719889.
01      FDV$_INC        PIC S9(9)  COMP GLOBAL  VALUE IS 2719897.
01      FDV$_MOD        PIC S9(9)  COMP GLOBAL  VALUE IS 2719905.
01      FDV$_IMP        PIC S9(9)  COMP GLOBAL  VALUE IS 2719922.
01      FDV$_FSP        PIC S9(9)  COMP GLOBAL  VALUE IS 2719930.
01      FDV$_IOL        PIC S9(9)  COMP GLOBAL  VALUE IS 2719938.
01      FDV$_FLB        PIC S9(9)  COMP GLOBAL  VALUE IS 2719946.
01      FDV$_ICH        PIC S9(9)  COMP GLOBAL  VALUE IS 2719954.
01      FDV$_FCH        PIC S9(9)  COMP GLOBAL  VALUE IS 2719962.
01      FDV$_FRM        PIC S9(9)  COMP GLOBAL  VALUE IS 2719970.
01      FDV$_FNM        PIC S9(9)  COMP GLOBAL  VALUE IS 2719978.
01      FDV$_LIN        PIC S9(9)  COMP GLOBAL  VALUE IS 2719986.
01      FDV$_FLD        PIC S9(9)  COMP GLOBAL  VALUE IS 2719994.
01      FDV$_NOF        PIC S9(9)  COMP GLOBAL  VALUE IS 2720002.
01      FDV$_DSP        PIC S9(9)  COMP GLOBAL  VALUE IS 2720010.
01      FDV$_NSC        PIC S9(9)  COMP GLOBAL  VALUE IS 2720018.
01      FDV$_DNM        PIC S9(9)  COMP GLOBAL  VALUE IS 2720026.
01      FDV$_DLN        PIC S9(9)  COMP GLOBAL  VALUE IS 2720034.
01      FDV$_UTR        PIC S9(9)  COMP GLOBAL  VALUE IS 2720042.
01      FDV$_IOR        PIC S9(9)  COMP GLOBAL  VALUE IS 2720050.
01      FDV$_IFN        PIC S9(9)  COMP GLOBAL  VALUE IS 2720058.
01      FDV$_ARG        PIC S9(9)  COMP GLOBAL  VALUE IS 2720066.
01      FDV$_INI        PIC S9(9)  COMP GLOBAL  VALUE IS 2720074.
01      FDV$_STR        PIC S9(9)  COMP GLOBAL  VALUE IS 2720082.
01      FDV$_IVM        PIC S9(9)  COMP GLOBAL  VALUE IS 2720090.
01      FDV$_FVM        PIC S9(9)  COMP GLOBAL  VALUE IS 2720098.
01      FDV$_ITT        PIC S9(9)  COMP GLOBAL  VALUE IS 2720106.
01      FDV$_TCA        PIC S9(9)  COMP GLOBAL  VALUE IS 2720114.
01      FDV$_STA        PIC S9(9)  COMP GLOBAL  VALUE IS 2720122.
01      FDV$_WID        PIC S9(9)  COMP GLOBAL  VALUE IS 2720130.
01      FDV$_NFL        PIC S9(9)  COMP GLOBAL  VALUE IS 2720138.
01      FDV$_IBF        PIC S9(9)  COMP GLOBAL  VALUE IS 2720146.
01      FDV$_NDS        PIC S9(9)  COMP GLOBAL  VALUE IS 2720154.
01      FDV$_UDP        PIC S9(9)  COMP GLOBAL  VALUE IS 2720162.
01      FDV$_UAR        PIC S9(9)  COMP GLOBAL  VALUE IS 2720170.
01      FDV$_UNF        PIC S9(9)  COMP GLOBAL  VALUE IS 2720178.
01      FDV$_CAN        PIC S9(9)  COMP GLOBAL  VALUE IS 2720194.
01      FDV$_KIF        PIC S9(9)  COMP GLOBAL  VALUE IS 2720202.
01      FDV$_KEX        PIC S9(9)  COMP GLOBAL  VALUE IS 2720210.
01      FDV$_KTW        PIC S9(9)  COMP GLOBAL  VALUE IS 2720218.
```

```
01      FDV$_KIL        PIC S9(9)  COMP GLOBAL  VALUE IS 2720226.
01      FDV$_TMO        PIC S9(9)  COMP GLOBAL  VALUE IS 2720234.
01      FDV$_LLI        PIC S9(9)  COMP GLOBAL  VALUE IS 2720242.
01      FDV$_VAL        PIC S9(9)  COMP GLOBAL  VALUE IS 2720250.
01      FDV$_IFU        PIC S9(9)  COMP GLOBAL  VALUE IS 2720258.
01      FDV$_SYS        PIC S9(9)  COMP GLOBAL  VALUE IS 2720266.

*****************************************************************
* FMS status codes returned when FDV$STAT routine is called.    *
*****************************************************************
* Success codes.

01      FDV$K_SUC       PIC S9(9)  COMP GLOBAL  VALUE IS       1.
01      FDV$K_INC       PIC S9(9)  COMP GLOBAL  VALUE IS       2.
01      FDV$K_MOD       PIC S9(9)  COMP GLOBAL  VALUE IS       3.

* Failure codes

01      FDV$K_IMP       PIC S9(9)  COMP GLOBAL  VALUE IS      -2.
01      FDV$K_FSP       PIC S9(9)  COMP GLOBAL  VALUE IS      -3.
01      FDV$K_IOL       PIC S9(9)  COMP GLOBAL  VALUE IS      -4.
01      FDV$K_FLB       PIC S9(9)  COMP GLOBAL  VALUE IS      -5.
01      FDV$K_ICH       PIC S9(9)  COMP GLOBAL  VALUE IS      -6.
01      FDV$K_FCH       PIC S9(9)  COMP GLOBAL  VALUE IS      -7.
01      FDV$K_FRM       PIC S9(9)  COMP GLOBAL  VALUE IS      -8.
01      FDV$K_FNM       PIC S9(9)  COMP GLOBAL  VALUE IS      -9.
01      FDV$K_LIN       PIC S9(9)  COMP GLOBAL  VALUE IS     -10.
01      FDV$K_FLD       PIC S9(9)  COMP GLOBAL  VALUE IS     -11.
01      FDV$K_NOF       PIC S9(9)  COMP GLOBAL  VALUE IS     -12.
01      FDV$K_DSP       PIC S9(9)  COMP GLOBAL  VALUE IS     -13.
01      FDV$K_NSC       PIC S9(9)  COMP GLOBAL  VALUE IS     -14.
01      FDV$K_DNM       PIC S9(9)  COMP GLOBAL  VALUE IS     -15.
01      FDV$K_DLN       PIC S9(9)  COMP GLOBAL  VALUE IS     -16.
01      FDV$K_UTR       PIC S9(9)  COMP GLOBAL  VALUE IS     -17.
01      FDV$K_IOR       PIC S9(9)  COMP GLOBAL  VALUE IS     -18.
01      FDV$K_IFN       PIC S9(9)  COMP GLOBAL  VALUE IS     -19.
01      FDV$K_ARG       PIC S9(9)  COMP GLOBAL  VALUE IS     -20.
01      FDV$K_INI       PIC S9(9)  COMP GLOBAL  VALUE IS     -21.
01      FDV$K_STR       PIC S9(9)  COMP GLOBAL  VALUE IS     -22.
01      FDV$K_FVM       PIC S9(9)  COMP GLOBAL  VALUE IS     -23.
01      FDV$K_IVM       PIC S9(9)  COMP GLOBAL  VALUE IS     -24.
01      FDV$K_ITT       PIC S9(9)  COMP GLOBAL  VALUE IS     -25.
01      FDV$K_TCA       PIC S9(9)  COMP GLOBAL  VALUE IS     -26.
01      FDV$K_STA       PIC S9(9)  COMP GLOBAL  VALUE IS     -27.
01      FDV$K_WID       PIC S9(9)  COMP GLOBAL  VALUE IS     -28.
01      FDV$K_NFL       PIC S9(9)  COMP GLOBAL  VALUE IS     -29.
01      FDV$K_IBF       PIC S9(9)  COMP GLOBAL  VALUE IS     -30.
01      FDV$K_NDS       PIC S9(9)  COMP GLOBAL  VALUE IS     -31.
01      FDV$K_UDP       PIC S9(9)  COMP GLOBAL  VALUE IS     -33.
01      FDV$K_UAR       PIC S9(9)  COMP GLOBAL  VALUE IS     -34.
01      FDV$K_UNF       PIC S9(9)  COMP GLOBAL  VALUE IS     -35.
01      FDV$K_CAN       PIC S9(9)  COMP GLOBAL  VALUE IS     -39.
01      FDV$K_KIF       PIC S9(9)  COMP GLOBAL  VALUE IS     -40.
```

```
01  FDV$K_KEX  PIC S9(9) COMP GLOBAL VALUE IS -41.
01  FDV$K_KTW  PIC S9(9) COMP GLOBAL VALUE IS -42.
01  FDV$K_KIL  PIC S9(9) COMP GLOBAL VALUE IS -43.
01  FDV$K_TMO  PIC S9(9) COMP GLOBAL VALUE IS -44.
01  FDV$K_LLI  PIC S9(9) COMP GLOBAL VALUE IS -45.
01  FDV$K_VAL  PIC S9(9) COMP GLOBAL VALUE IS -47.
01  FDV$K_IFU  PIC S9(9) COMP GLOBAL VALUE IS -48.
01  FDV$K_SYS  PIC S9(9) COMP GLOBAL VALUE IS -49.
```

Chapter 6
Programming FMS Applications in VAX–11 FORTRAN

The FMS Form Driver processes all Form Driver calls according to VAX–11 standards and language-specific rules. These rules define how arguments are passed to the Form Driver and how values are returned to your program. Language-specific information is briefly presented in this manual. For more detail, refer to the VAX–11 FORTRAN document set.

Your VAX–11 FORTRAN application program must comply with the requirements of the VAX–11 FORTRAN FMS interface. Topics discussed in this chapter include:

- Form Driver Routines
  - Invoking Form Driver Routines as Subroutines
  - Accessing Form Driver Status Codes as Functions
- Argument Passing in FMS
- Null Arguments
- FMS Data Types
  - Character Strings
  - Longword Binary Integers
  - Word Binary Integers
- Non-FMS Data Types
- One-Dimensional Arrays
- Allocation: Workspace, Terminal Control Area, and Run-Time Memory-Resident Form Area
- Precautions for Using FMS
- Data Conversion
- Sample Application Program in VAX–11 FORTRAN

A sample program written in FORTRAN (SAMPFOR.FOR) appears at the end of this chapter. Following the code for SAMPFOR.FOR are Form Driver definition files created for SAMPFOR.FOR. Command file information needed to build the Sample Application program is in Section 6.10.2.

Examples from the Sample Application are used throughout the text to illustrate language issues. Where appropriate examples from SAMPFOR.FOR do not exist, other examples are provided.

6.1 Form Driver Routines

You can call any FMS routine as a subroutine or as a function. Syntax follows standard VAX–11 FORTRAN requirements.

6.1.1 Invoking Form Driver Routines as Subroutines

You use the procedure call statement to invoke an FMS Form Driver routine. For example:

```
CALL FDV$WAIT ()
```

Calls the Form Driver routine FDV$WAIT and passes no arguments.

```
CALL FDV$GET (OPTION, TERMINATOR, 'OPTION')
```

Calls the Form Driver routine FDV$GET and passes three arguments.

See Appendix A for a complete list of Form Driver calls. The calling sequence for each Form Driver routine, data access codes, data types, and passing mechanisms are presented in language-independent notation as specified by the VAX–11 Procedure Calling and Condition Handling Standard. For further detail about the VAX–11 Procedure Calling and Condition Handling Standard, refer to the *VAX–11 Run-Time Library Reference Manual.*

6.1.2 Accessing Form Driver Status Codes as Functions

An FMS status code is returned to the calling program at the completion of all Form Driver calls. To receive the returned status code from a Form Driver routine, you activate the routine with a function reference rather than with a call statement. Note that this returns a standard VMS status code. For portability, other status mechanisms can also be used. (For more information, see the *VAX–11 FMS Form Driver Reference Manual*, Chapter 2.)

Before you reference a status_return function, you declare its data type to be INTEGER*4. The following statements declare and call FDV$GET as an FMS function:

```
INTEGER*4 FDV$GET
INTEGER*4 RETURN_STATUS

RETURN_STATUS = FDV$GET (OPTION, TERMINATOR, 'OPTION')
```

Alternatively, you can implicitly declare the data type of all FMS function names, using the IMPLICIT statement. The declaration IMPLICIT INTEGER*4 F declares the data type of all the Form Driver subroutines to be INTEGER*4 since all FMS-related names begin with F. Note that you cannot use this method if you are using the IMPLICIT NONE statement to ensure explicit declaration of all names in your program.

6.2 Argument Passing in FMS

The argument passing mechanism refers to the way in which data is passed to a called routine. The VAX–11 Procedure Calling Standard has three methods for passing arguments:

- By reference
- By descriptor
- By value

FMS routines, however, expect arguments to be passed only by reference and by descriptor.

By reference specifies that the storage location of the argument is passed to the routine. FMS expects integers to be passed by reference, which is also the FORTRAN default passing mechanism for integers.

By descriptor specifies that the address of a descriptor data structure is passed to the routine. FMS expects character strings and integer arrays to be passed by descriptor, which is the FORTRAN default passing mechanism for character strings (but not integer arrays).

Integer arrays require special treatment. Although the FORTRAN default passing mechanism for integer arrays is by reference, FMS has built-in functions for passing arguments when you wish to override FORTRAN default mechanisms. In this case, you can use the %DESCR function to force the argument list entry to use the descriptor mechanism. For example:

```
INTEGER*4 WORKSPACE (3)
CALL FDV$AWKSP (%DESCR(WORKSPACE), 2000)
```

6.3 Null Arguments

When the call syntax includes optional arguments and you do not wish to specify all of the information, you can use null arguments. Any optional argument can be omitted to simplify your program. A comma functions as a placeholder for each null argument. Optional arguments to the right of the last specified argument can simply be omitted from the call. In the following example, the FDV$GETAL call passes only the field terminator value:

```
CALL FDV$GETAL (, FLDTRM)
```

6.4 FMS Data Types

6.4.1 Character Strings

The character string is one of the general data types used by FMS. For example, the FDV$GET call passes the character strings for field value (OPTION) and field name ('OPTION'):

```
CALL FDV$GET (OPTION, TERMINATOR, 'OPTION')
```

You must be certain that your strings are initially declared to be long enough to accommodate your FMS data. One option is to declare your fixed-length strings to be the exact length of the FMS data to be returned. You can use the FMS/DESCRIPTION/DECLARATIONS command to get the length of the strings.

Alternatively, a single string variable can be used in different FMS calls to transfer data to or from several forms and fields. You must declare the string variable to be at least as large as the longest field value string that will be returned to your program. You can use the FDV$RETLE call to return the length of the valid portion of the field value in the string variable. This length can then be used when referencing the data that has been entered in the field. For example:

```
CHARACTER*100 ACCOUNT
 .
 .
 .
CALL FDV$GET (ACCOUNT, TERMINATOR, 'FIELD')
CALL FDV$RETLE (LENGTHFIELD, 'FIELD')
 .
 .
 .
WRITE (10,*) ACCOUNT(:LENGTHFIELD)
```

After the execution of the FDV$RETLE call, LENGTHFIELD is equal to the length of the field named 'FIELD.' It is also equal to the valid portion of the string that is defined by the string descriptor ACCOUNT. LENGTHFIELD can now be used to reference the data that was entered in the field named 'FIELD', and that is now in the variable ACCOUNT. If you do not use the substring specifier (:LENGTHFIELD) when referencing ACCOUNT, you will reference the entire variable, including any blanks used by the Form Driver to pad the string.

A useful application of the FDV$RETLE call is in general purpose user action routines.

6.4.2 Longword Binary Integers

The longword binary integer is another general data type used by FMS. For example, the FDV$ATERM call passes the longword value for terminal control area (12) and logical I/O channel number (2):

```
CALL FDV$ATERM (%DESCR(TCA),12,2)
```

Numeric arguments must be longword binary integers (INTEGER*4). If you pass other numeric types to the Form Driver, the calls do not work properly. An exception is the FDV$DFKBD call (see next section).

Note that you can declare numeric arguments to be INTEGER because the VAX default is INTEGER*4. This will increase the compatibility of your program with PDP–11s, which have a default of INTEGER*2.

6.4.3 Word Binary Integers

The defkbd argument is a word integer array passed when the FDV$DFKBD routine is called. FMS expects arrays to be passed by descriptor.

6.5 Non-FMS Data Types

FORTRAN data types that are not recognized by FMS can be used in your FORTRAN application program provided they are not passed to the Form Driver.

6.6 One-Dimensional Arrays

One-dimensional arrays are structures that can be used in FMS for the following arguments:

- tca (terminal control area)
- wksp (workspace)
- mloc (memory location)
- defkbd (define keyboard)

You must provide FMS with storage space for these arguments. You can do that by defining them to be:

- INTEGER*4 arrays or character strings for tca, wksp, and mloc
- word integer arrays for defkbd

In the Sample Application program, the tca, wksp, and mloc arguments are passed to several Form Driver routines. These arguments are defined as integer array variables. The following declarations establish names and storage for the integer array variables WORKSPACE, CHECKWKSP, TCA, and MENU_FORM:

```
C  Data definitions

   INTEGER   WORKSPACE (3),    !General workspace
   1         CHECKWKSP (3),    !Check workspace
   2         TCA       (3),    !Terminal Control Area
   3         MENU_FORM (500),  !Storage for memory-resident form
```

You may alternatively declare these variables to be character strings in your own application program. You could then take advantage of FORTRAN's default passing mechanism for character strings (by descriptor). This would avoid the need to use %DESC to force the descriptor mechanism for passing integer arrays. Furthermore, you could then declare the actual storage area of character strings in bytes. The following statements declare and allocate space to the character strings WORKSPACE, CHECKWKSP, and TCA:

```
CHARACTER*12 WORKSPACE
CHARACTER*12 CHECKWKSP
CHARACTER*12 TCA
```

6.7 Allocation: Workspace, Terminal Control Area, and Run-Time Memory-Resident Form Area

The FMS workspace, terminal control area, and run-time memory-resident form area variables are character strings or longword integer arrays. These variables should be placed in a common storage area of your program. Note that this is not done in the Sample Application program. The sample program's structure protects the workspaces, terminal control areas, and run-time memory-resident form areas implicitly.

The Form Driver needs 12 bytes to associate user information about each workspace and terminal control area. The allocation for a run-time memory-resident form area must be the size of the form. The space for all of these variables is allocated by your application program. Note that FMS uses *only* 12 bytes of space for the workspace and terminal control area allocation. Any allocation greater than 12 bytes is wasted. You declare your workspace, terminal control area, and run-time memory-resident form area only once. FMS remembers their addresses after the addresses have been initially passed to the Form Driver.

For each workspace, the Form Driver also allocates an additional amount of storage space based on your estimate of the amount of memory needed to store your largest form. If your estimate is too small, the Form Driver allocates more space automatically, but performance may be affected. An adequate estimate results in more efficient operation of the Form Driver. You can use the FMS/DIRECTORY/FULL command to find out how much space to allocate.

In the following example from the Sample Application program, workspace is allocated and the FDV$AWKSP routine is called. When the FDV$AWKSP routine is called, the first argument (WORKSPACE) specifies the area of memory to be used for your workspace. The second argument specifies an estimate of the workspace size that you will need to display the largest form in your application.

```
C   Data definitions
    INTEGER   WORKSPACE (3),        !General workspace
    1         CHECKWKSP (3),        !Check workspace
    CALL FDV$AWKSP (%DESCR(CHECKWKSP), 2000)
    CALL FDV$AWKSP (%DESCR(WORKSPACE), 2000)
```

6.8 Precautions for Using FMS

6.8.1 Memory Areas Used Exclusively by FMS

The locations for terminal control area, workspace, and run-time memory-resident form area are used exclusively by FMS. The terminal control area and workspace are attached with the FDV$ATERM and FDV$AWKSP calls and remain allocated until the FDV$DTERM and FDV$DWKSP calls are issued or until the program ends. The run-time memory-resident form area, used in the FDV$READ call, remains allocated until the FDV$DEL call is issued or until the program ends. You never touch the terminal control areas, workspaces, or run-time memory-resident form areas in your program except to pass their addresses to the Form Driver.

6.8.2 Why You Should Use the COMMON Attribute

Parameters to the following Form Driver routines should be used with caution:

FDV$ATERM	Attach terminal
FDV$AWKSP	Attach form workspace
FDV$READ	Read form into memory
FDV$SSRV	Specify status reporting variables

For example, once an FDV$SSRV call is issued, the program variables that contain the FMS status and RMS status become volatile and can change at any call point. As a general rule, you should place the status reporting variables in COMMON.

In cases where you need both the FMS and RMS statuses, the FDV$STAT routine can be used. Note that only the FDV$STAT and FDV$SSRV calls provide RMS status. With the FDV$STAT routine, you do not have to worry about volatility.

The locations for terminal control area, workspace, run-time memory-resident form area, and status reporting variables must all continue to exist while the Form Driver is using them. They must remain allocated until the terminal control area and workspace are detached, until forms in memory location are deleted, and until the status reporting variables are no longer used. The variables can be protected by placing them in a common storage area; otherwise, the compiler might place them in dynamic storage or reuse their storage area.

6.9 Data Conversion

FMS uses only ASCII character strings to display data. All information displayed on the terminal screen, and all information received from the terminal operator, is represented as ASCII field values. Any manipulation of numeric data requires conversion of ASCII character strings to numeric data, and conversion of numeric data back to ASCII character strings. In

the following discussion of conversion routines, you should assume that the receiving data type can support the largest number that is likely to be generated.

In the Sample Application, the following steps are taken to get a new account balance after writing a check.

```
CALL FDV$RET (RI_AMTPAY, 'AMTPAY')
READ (RI_AMTPAY, '(I6)') AMTPAY
BALANCE = BALANCE - AMTPAY
TOTPAY = TOTPAY + AMTPAY

WRITE (RI_BALANCE, '(I6)') BALANCE
CALL FDV$PUT (RI_BALANCE, BALANCE')
```

In this example, the FORTRAN internal file READ converts the character variable RI_AMTPAY to the integer variable AMTPAY according to the format specification of I6. The integer value of the variable AMTPAY is subtracted from the integer value of the variable BALANCE to produce a new value for BALANCE. When converting ASCII to numeric, your application is assured a successful conversion if the field on the form was defined as numeric. This eliminates the need for an "ERR =" clause on the READ statement.

After the data operations have been completed, the FORTRAN internal file WRITE converts the integer variable BALANCE to a character expression RI_BALANCE. The value for the balance is displayed in the right-justified field 'BALANCE'. The rightmost digit from the program is displayed in the field's rightmost character position. The remaining digits of the character expression are placed to the left of the rightmost digit. If output is longer than the field, FMS truncates on the left. (The Form Driver displays a data length error message (FDV$_DLN) only if you have set FMS Debug mode.)

For other data conversion options, see the general conversion routines in the *VAX–11 Run-Time Library Reference Manual.*

6.10 Sample Application Program in VAX–11 FORTRAN

The FMS Sample Application program (SAMPFOR.FOR) is part of the FMS distribution kit. When FMS is installed, SAMPFOR.FOR is placed in the directory FMS$EXAMPLES. Designed to be a demonstration program and learning tool, SAMPFOR.FOR shows most of the features provided by FMS. The entire Sample Application program appears at the end of this chapter.

6.10.1 Form Driver Definition Files

The files FDVDEF.FOR, SMPACCOM.FOR, SMPREGCOM.FOR, SMPSTATUS.FOR, and SMPWORK.FOR are part of the Sample Application program package. When FMS is installed, these files are placed in the directory FMS$EXAMPLES. They appear after the Sample Application source code.

FDVDEF.FOR contains a variety of codes for the Form Driver routines used in the Sample Application program. Although these codes have been created for use in SAMP.BAS, they can provide you with a helpful starting point as you create definitions for your own application program. The file FDVDEF.FOR includes:

- FMS terminator codes
- Function key terminators returned from the FDV$GET and FDV$WAIT calls
- Form Driver key ffunctions for use with the FDV$DFKBD call
- User action routine (UAR) return codes, which are returned by the UARs to the Form Driver:

  Field completion UAR return codes

  Help UAR return codes

  Function key UAR return codes
- VMS status codes returned when Form Driver routines are called as functions. These codes can be signaled.
- FMS status codes returned when the FDV$STAT routine is called as a function
- Declarations of Form Driver routines

6.10.2 Command File for Building the Sample Application Program

The command file for building the Sample Application program includes all the information that you need to compile and link SAMPFOR.FOR. When FMS is installed, the command file is placed in the directory FMS$EXAMPLES.

```
$!   S A M P F O R . C O M
$!
$!   Compile and link the FORTRAN version of the FMS V2 Sample Application
$!
$!   The FORTRAN source files are:    SAMPFOR.FOR
$!                                    FDVDEF.FOR
$!                                    SMPACCOM.FOR
$!                                    SMPREGCOM.FOR
$!                                    SMPSTATUS.FOR
$!                                    SMPWORK.FOR
$!
$!   SMPVECTOR.OBJ and SMPMEMRES.OBJ were produced by the FMS commands:
$!
$!      $ FMS/VECTOR/OUTPUT=SMPVECTOR SAMP.FLB
$!      $ FMS/MEMORY/OUTPUT=SMPMEMRES SAMP.FLB/FORM=(HELP_KEYS,HELP_MENU)
$!
$ LIBRARY/CREATE/TEXT  SMPFORTXT      SMPACCOM.FOR /MODULE=ACCOUNT_COMMON,-
                       SMPREGCOM.FOR /MODULE=REGISTER_COMMON,-
                       SMPSTATUS.FOR /MODULE=STATUS_AREA,-
                       SMPWORK.FOR /MODULE=WORK_AREA
$!
$ FORTRAN              SAMPFOR
$ LINK                 SAMPFOR, FMS$EXAMPLES:SMPVECTOR, FMS$EXAMPLES:SMPMEMRES
```

```
      PROGRAM SAMPLE
      IMPLICIT NONE

C     SAMP -- The FMS Sample Application Program
C
C Data definitions
C
C FMS related
C


C
CC    in the text module WORK_AREA in library SMPFORTXT, there are some
C     variables the declaration of which should be verified:
C
CC    INTEGER  WORKSPACE( 3 ),        !General workspace
C     1        CHECKWKSP( 3 ),        !Check workspace
C     2        TCA( 3 ),              !Terminal Control Area
C     3        MENU_FORM(500),        !Storage for memory resident form
C     4        CHECK_FORM(750),       !Storage for memory resident form
C     5        DPOSIT_FORM(500),      !Storage for memory resident form
C
      INCLUDE 'SMPFORTXT(WORK_AREA)'
      INCLUDE 'SMPFORTXT(STATUS_AREA)'
C
C     Money.
C     Note that all money is kept internally as integers (in cents).
C     It is only when the quantities are output that they look like
C     dollars, since all the money fields have periods as field
C     markers in the right places and they are right justified or
C     fixed decimal.
C
C     Register data.
C     It would be most convenient to be able to define an array
C     of structures for the register, but it can't be done
C     in FORTRAN (it can be done for some other languages).  What's one
C     instead is to define a single structure into which to put data via
C     structure names and also an array of strings.  After data has been
C     be put into the structure, it is copied to the array for convenience
C     in scrolling.
```

```
C
C       Initialize FMS
C       Attach default terminal
C       Attach normal and check workspaces (order important for help
C         and refresh during CHECK/CHECK_DONE time--try switching and see).
C       Open form library, attach to channel 1
C       Set keypad mode to application
C       Set signal mode to bell (default, but it's fun to do)
C


        CALL FDV$ATERM( %DESCR(TCA),12 ,2 )
        CALL GET_AND_CHECK_FMSSTATUS
        CALL FDV$AWKSP( %DESCR(CHECKWKSP), 2000 )
        CALL GET_AND_CHECK_FMSSTATUS()
        CALL FDV$AWKSP( %DESCR(WORKSPACE), 2000 )
        CALL GET_AND_CHECK_FMSSTATUS()
        CALL FDV$LOPEN( 'FMS$EXAMPLES:SAMP', 1 )
        CALL GET_AND_CHECK_FMSSTATUS()
        CALL FDV$SPADA( 1 )
        CALL FDV$SSIGQ( 0 )
C
C       Set all future calls to return status to the two status recording
C       variables FMSSTATUS and RMSSTATUS without having to call the
C       the FDV$STAT routine.
C
        CALL FDV$SSRV( FMSSTATUS, RMSSTATUS )
C
C       Read in a few forms from the form library onto the dynamic
C       resident form list.  You may be able to detect the difference
C       in the form to form access times for those forms which have to be
C       accessed from the form library on disk and those forms which are
C       on the dynamic or static memory resident form list.  See the
C       installation notes for this program (the LINK command) to see
C       which forms are on the static memory resident form list.
C
        CALL FDV$READ( 'MENU', %DESCR(MENU_FORM), 2000, SIZE_MENU)
        CALL FDV$READ( 'CHECK', %DESCR(CHECK_FORM), 3000, SIZE_CHECK)
        CALL FDV$READ( 'DEPOSIT', %DESCR(DPOSIT_FORM), 2000, SIZE_DPOSIT)
```

```
C
C       Initialize account information
C
        CALL INIT_ACCOUNT()
C
C        Put up welcome form, wait for response
C
        CALL FDV$CDISP( 'WELCOME' )
        CALL CHECK_FMSSTATUS()

        CALL FDV$WAIT
C
C        Process all menu requests
C

        CALL MENU()

C
C        Clean up and leave:
C          Close form library.
C          Reset keypad to numeric.
C          Delete a form from dynamic mem. res. form list just to show how.
C          Detach workspaces (not really necessary since DTERM would do it).
C          Detach terminal.
C
        CALL FDV$LCLOS
        CALL FDV$SPADA( 0 )
        CALL FDV$DEL( 'MENU' )
        CALL FDV$DWKSP( %DESCR(WORKSPACE) )
        CALL FDV$DWKSP( %DESCR(CHECKWKSP) )
        CALL FDV$DTERM( %DESCR(TCA))
        STOP
        END
```

```
      SUBROUTINE INIT_ACCOUNT
C
C             Read from file SAMP.DAT into internal variables.
C             Set up the workspace for checks and fill in the check form
C             with the account's name, address, and account number.
C
      IMPLICIT NONE

      INCLUDE 'SMPFORTXT(REGISTER_COMMON)'
      INCLUDE 'SMPFORTXT(ACCOUNT_COMMON)'
C
C      Open file, set account data
C
      OPEN(UNIT=5, STATUS='OLD', READONLY, FILE='FMS$EXAMPLES:SAMP.DAT')
      READ (UNIT=5, FMT=10, END=100) ACCOUNT
10    FORMAT(A)

C      Read the remaining records into the check register, counting them.
C      The last record has the current balance, and some record has the
C      last check number used (not necessarily the last record).
C      Note that in FORTRAN the record is read into the array and reference
C      to the check number is via a substring rather than symbolically.
C      Other languages may access differently.

      LASTCHNUM = 0
      LASTREGNUM = 0
      DO WHILE (LASTREGNUM .LT.  REGSIZE)
          READ (UNIT=5, FMT=20, END=100) REGARRAY(LASTREGNUM+1)
20        FORMAT(A)
          LASTREGNUM = LASTREGNUM + 1
          IF ( REGARRAY(LASTREGNUM)(1:4) .NE. '    ' ) THEN
              READ(REGARRAY(LASTREGNUM)(1:4),'(I4)') LASTCHNUM
          ENDIF
      ENDDO

C      Reached here without hitting end of file, should probably print
C      message or something, except that this is just a l'il ol' demo.
C      As it is, just fall through and ignore remaining records.

100   CLOSE(5)
```

```
C        Reach here as result of end of file--last record tried didn't read.
C        Check for data file in error.
C        Take balance from last record read.
C        Set session sums to zero to say no activity yet.

         IF (LASTREGNUM .EQ. 0) THEN
             PRINT *,'DATA FILE IN ERROR'
             STOP
         ENDIF

         READ(REGARRAY(LASTREGNUM)(59:64),'(I6)') BALANCE
         SBALANCE = BALANCE
         TOTDEP = 0
         TOTPAY = 0

C        Set up the check workspace once so we don't have to do it every time.

         CALL FORMAT_CHECK()
         RETURN
         END




         SUBROUTINE FORMAT_CHECK
         INTEGER TRIM_LENGTH_FIRST, TRIM_LENGTH_CITY
C
C                Format account data onto check form in the check workspace.
C
         INCLUDE 'SMPFORTXT(ACCOUNT_COMMON)'
         INCLUDE 'SMPFORTXT(WORK_AREA)'
C
C                Call the system routine STR$TRIM to trim trailing blanks
C                from the first name and the city name.
C
         CALL STR$TRIM (TRIM_FIRST, FIRST, TRIM_LENGTH_FIRST)
         CALL STR$TRIM (TRIM_CITY, CITY, TRIM_LENGTH_CITY)
```

```
C
C           Format the check.
C
      CALL FDV$SWKSP( %DESCR(CHECKWKSP) )
      CALL FDV$LOAD( 'CHECK' )
      CALL FDV$PUT(
     1        TRIM_FIRST(1:TRIM_LENGTH_FIRST) // ' ' //
     1        MIDDLE(1:1) // '. ' // LAST, 'NAME' )
      CALL FDV$PUT( STREET, 'STREET' )
      CALL FDV$PUT( TRIM_CITY(1:TRIM_LENGTH_CITY) // ', '
     1        // STATE // ' ' // ZIP , 'CSZ' )
      CALL FDV$PUT( HOMEPH, 'HOMEPH' )
      CALL FDV$PUT( ACCTNO, 'ACCTNO' )
      CALL FDV$SWKSP( %DESCR(WORKSPACE) )
      RETURN
      END



      SUBROUTINE MENU
C
C           Accept inputs from the menu form and dispatch to the
C           appropriate routine.  Repeat until option 1 (exit) is
C           chosen.  The UARs in the form guarantee that we get back
C           only inputs '1'-'5' with the correct terminators.
C           Options are:
C                   1 => Exit
C                   2 => Write checks
C                   3 => Make deposit
C                   4 => View register
C                   5 => View account data
C
      IMPLICIT NONE
      INCLUDE 'SMPFORTXT(WORK_AREA)'

      CHARACTER*1 OPTION
      INTEGER TRANSFER_CONTROL
      OPTION = ' '

      DO WHILE (OPTION .NE. '1')
          CALL FDV$CDISP( 'MENU' )
```

```
            CALL CHECK_FMSSTATUS()
            CALL FDV$GET( OPTION, TERMINATOR, 'OPTION' )
            CALL CHECK_FMSSTATUS()

            READ (OPTION,'(I1)') TRANSFER_CONTROL
            GOTO (100,20,30,40,50) TRANSFER_CONTROL

C
20          CALL WRITE_CHECK()
            GOTO 100
C
30          CALL MAKE_DEPOSIT()
            GOTO 100
C
40          CALL VIEW_REGISTER()
            GOTO 100
C
50          CALL VIEW_ACCOUNT_DATA()

100     ENDDO
        RETURN
        END



        SUBROUTINE WRITE_CHECK
C
C               Write one or more checks
C
        IMPLICIT NONE

        INCLUDE 'FDVDEF'
        INCLUDE 'SMPFORTXT(REGISTER_COMMON)'
        INCLUDE 'SMPFORTXT(WORK_AREA)'

        CHARACTER*6 BALANCE_STRING

C        Turn on LED 3 on the VT100 during this routine, just to show how.

        CALL FDV$LEDON( 3 )
```

```
C
C               Format the check.
C
        CALL FDV$SWKSP( %DESCR(CHECKWKSP) )
        CALL FDV$LOAD( 'CHECK' )
        CALL FDV$PUT(
        1       TRIM_FIRST(1:TRIM_LENGTH_FIRST) // ' ' //
        1       MIDDLE(1:1) // '. ' // LAST, 'NAME' )
        CALL FDV$PUT( STREET, 'STREET' )
        CALL FDV$PUT( TRIM_CITY(1:TRIM_LENGTH_CITY) // ', '
        1       // STATE // ' ' // ZIP , 'CSZ' )
        CALL FDV$PUT( HOMEPH, 'HOMEPH' )
        CALL FDV$PUT( ACCTNO, 'ACCTNO' )
        CALL FDV$SWKSP( %DESCR(WORKSPACE) )
        RETURN
        END



        SUBROUTINE MENU
C
C       Accept inputs from the menu form and dispatch to the
C       appropriate routine.  Repeat until option 1 (exit) is
C       chosen.  The UARs in the form guarantee that we get back
C       only inputs '1'-'5' with the correct terminators.
C       Options are:
C               1 => Exit
C               2 => Write checks
C               3 => Make deposit
C               4 => View register
C               5 => View account data
C
        IMPLICIT NONE
        INCLUDE 'SMPFORTXT(WORK_AREA)'

        CHARACTER*1 OPTION
        INTEGER TRANSFER_CONTROL
        OPTION = ' '

        DO WHILE (OPTION .NE. '1')
            CALL FDV$CDISP( 'MENU' )
```

```
            CALL CHECK_FMSSTATUS()
            CALL FDV$GET( OPTION, TERMINATOR, 'OPTION' )
            CALL CHECK_FMSSTATUS()

            READ (OPTION,'(I1)') TRANSFER_CONTROL
            GOTO (100,20,30,40,50) TRANSFER_CONTROL

C
20          CALL WRITE_CHECK()
            GOTO 100
C
30          CALL MAKE_DEPOSIT()
            GOTO 100
C
40          CALL VIEW_REGISTER()
            GOTO 100
C
50          CALL VIEW_ACCOUNT_DATA()

100     ENDDO
        RETURN
        END



        SUBROUTINE WRITE_CHECK
C
C               Write one or more checks
C
        IMPLICIT NONE

        INCLUDE 'FDVDEF'
        INCLUDE 'SMPFORTXT(REGISTER_COMMON)'
        INCLUDE 'SMPFORTXT(WORK_AREA)'

        CHARACTER*6 BALANCE_STRING

C        Turn on LED 3 on the VT100 during this routine, just to show how.

        CALL FDV$LEDON( 3 )
```

```
C        Mark WORKSPACE not displayed so it doesn't show up during a refresh
C        Put up CHECK form from already loaded workspace
C        and display current balance

         CALL FDV$NDISP
         CALL FDV$SWKSP( %DESCR(CHECKWKSP) )
         CALL FDV$DISPW

         WRITE(BALANCE_STRING, '(I6)') BALANCE
         CALL FDV$PUT( BALANCE_STRING, 'BALANCE' )

C        Process checks until a keypad period is read

         TERMINATOR = 0
         DO WHILE (TERMINATOR .NE. FDV$K_KP_PER)
             CALL PROCESS_ONE_CHECK()
             CALL GIVE_CONTINUE_OPTIONS()
         ENDDO

C        Turn off LED 3 on VT100

         CALL FDV$LEDOF( 3 )
         CALL FDV$SWKSP( %DESCR(WORKSPACE) )
         RETURN
         END



         SUBROUTINE PROCESS_ONE_CHECK

C        If input is terminated by kpd period, return with no action
C        Else deduct from balance and enter into register.
C        Note that a UAR in the form guarantees that the amount of
C        the check is always less than or equal to the balance.
C        Note that the form function key UAR  allows only kpd period
C        as terminator (other than FDV$K_FT_NTR).

         IMPLICIT NONE

         INCLUDE 'FDVDEF'
         INCLUDE 'SMPFORTXT(REGISTER_COMMON)'
         INCLUDE 'SMPFORTXT(WORK_AREA)'
```

```
      CHARACTER*4 CHECK_NUM_STRING

      CHARACTER*100   JUNK


      INTEGER AMTPAY

      WRITE(CHECK_NUM_STRING, '(I4)') LASTCHNUM+1
      CALL FDV$PUT( CHECK_NUM_STRING, 'NUMBER' )

      CALL FDV$GETAL( JUNK, TERMINATOR )
      IF (TERMINATOR .EQ. FDV$K_KP_PER) RETURN

C      If the check wouldn't fit in the register, don't process, just
C      give error message, wait for acknowledgement, and return

      IF (LASTREGNUM .EQ. REGSIZE) THEN
          CALL FDV$PUTL( 'Register full, can''t enter check' )
          CALL FDV$WAIT
          RETURN
      ENDIF

C      Get amount from check.
C      Update balance (in memory and on screen) and session sums.
C      Transfer form values to register item.

      CALL FDV$RET( RI_AMTPAY, 'AMTPAY' )
      READ(RI_AMTPAY,'(I6)') AMTPAY
      BALANCE = BALANCE - AMTPAY
      TOTPAY  = TOTPAY  + AMTPAY

      WRITE(RI_BALANCE, '(I6)') BALANCE
      CALL FDV$PUT( RI_BALANCE, 'BALANCE' )

      RI_AMTDEP = ' '
      CALL FDV$RET( RI_NUM, 'NUMBER' )
      CALL FDV$RET( RI_DATE, 'DATE' )
      CALL FDV$RET( RI_MEMPAYTO, 'PAYTO' )     ! Note: not from check's MEMO

C      Update register array and counters
C      (Note that the two step update (form->regitem->regarray)
C      is necessary in FORTRAN not necessarily in every language).
```

```
            LASTREGNUM = LASTREGNUM + 1
            LASTCHNUM  = LASTCHNUM  + 1
            REGARRAY( LASTREGNUM ) = REGITEM
            RETURN
            END



            SUBROUTINE GIVE_CONTINUE_OPTIONS

C           Finish off check processing by giving operator
C           three options:
C                   RETURN  Write another check
C                   KPD 0   Print the check into file SAMPCH.DAT
C                   KPD .   Return to menu
C           Check to see if check write was aborted by kpd per.
C           If so, then don't give any further choice, just abort.
C           Note that form function key UAR allows only the above
C           terminators to get through.

            IMPLICIT NONE

            INCLUDE 'FDVDEF'
            INCLUDE 'SMPFORTXT(WORK_AREA)'

            IF (TERMINATOR .EQ. FDV$K_KP_PER) RETURN

C            Tell the operator that the check has been paid by overlaying with
C            a new form, using the normal workspace, thereby saving the check
C            workspace in case another check is to be written.

            CALL FDV$SWKSP( %DESCR(WORKSPACE) )
            CALL FDV$DISP( 'CHECK_DONE' )
            CALL CHECK_FMSSTATUS()

C            Wait for operator to enter either KPD period, NTR, or KPD zero.
C            Print the check as many times as requested.
C            (Note that a UAR on the form guarantees that only those terminators
C            are accepted).
C            Process accordingly.
```

```
      CALL FDV$WAIT( TERMINATOR )
      DO WHILE (TERMINATOR .EQ. FDV$K_KP_0)
          CALL PRINT_THE_CHECK()
          CALL FDV$WAIT( TERMINATOR )
      ENDDO

C      If the choice is to quit,
C      then mark check wksp undisplayed so it doesn't appear during refresh,
C      else mark normal workspace (occupied by CHECK_DONE form) undisplayed
C        so it doesn't show during refresh and then clear its lines.
C        (Clearing the space occupied by the CHECK_DONE form, lines 20-23
C        is better donee by overlaying with a blank form to
C        aviod having to know the line numbers to clear

      IF (TERMINATOR .EQ. FDV$K_KP_PER) THEN
          CALL FDV$SWKSP( %DESCR(CHECKWKSP) )
          CALL FDV$NDISP
      ELSE
          CALL FDV$NDISP
          CALL FDV$CLEAR( 20, 4 )
          CALL FDV$SWKSP( %DESCR(CHECKWKSP) )
      ENDIF

C       Going to write another check now or eventually, so:

      CALL FDV$PUTD( 'AMTPAY' )
      CALL FDV$PUTD( 'MEMO' )
      CALL FDV$PUTD( 'PAYTO' )
      RETURN
      END



      SUBROUTINE PRINT_THE_CHECK

C             Print the check into the file SAMPCH.DAT
C             Use the check workspace, then switch back to the normal wksp
C             to keep things clean.

      IMPLICIT NONE
      INCLUDE 'SMPFORTXT(WORK_AREA)'
```

```
      CHARACTER*80    LINE
      CHARACTER*2     FIRSTL,
      1               LASTL

      INTEGER         FIRST_LINE_NUMBER,
      1               LAST_LINE_NUMBER,
      2               I,
      3               LINELENGTH

C      Open check writing file.  Note there's a new version for every check.
C      Switch workspaces
C
      OPEN(UNIT=2, FILE='SAMPCH.DAT', STATUS='NEW', CARRIAGECONTROL='LIST',
      1       RECORDSIZE=80 )
      CALL FDV$SWKSP( %DESCR(CHECKWKSP) )

C      Get the top and bottom lines of the check from the named data
C      (first two characters).

      CALL FDV$RETDN( 'FIRST', FIRSTL )
      CALL CHECK_FMSSTATUS()
      CALL FDV$RETDN( 'LAST', LASTL )
      CALL CHECK_FMSSTATUS()

C      Get lines from form.
C      Convert to line printer style.
C      Write to file.

      READ (FIRSTL, '(I2)') FIRST_LINE_NUMBER
      READ (LASTL,  '(I2)') LAST_LINE_NUMBER

      DO I = FIRST_LINE_NUMBER, LAST_LINE_NUMBER
          CALL FDV$RETFL( I, LINE, LINELENGTH )
          WRITE(2,'(A)') LINE(1:LINELENGTH)           !## , 1, LINELENGTH
      ENDDO
      CALL FDV$PUTL( 'Check written to file' )
      CLOSE (2)
      CALL FDV$SWKSP( %DESCR(WORKSPACE) )
      RETURN
      END
```

```
      SUBROUTINE MAKE_DEPOSIT

C             Make a deposit, enter into check register
C             Cancel on Keypad Period.
C             Note that the Form Function Key UAR allows only Kpd Period.
C
C      Put up deposit form with current balance

      IMPLICIT NONE

      INCLUDE 'FDVDEF'
      INCLUDE 'SMPFORTXT(REGISTER_COMMON)'
      INCLUDE 'SMPFORTXT(WORK_AREA)'

C      Deposit data (Read via FDV$GETAL)

      CHARACTER*60    DEPOSIT
      CHARACTER*7     DEP_DATE
      EQUIVALENCE (DEPOSIT(1:), DEP_DATE)
      CHARACTER*6     DEP_CURBAL
      EQUIVALENCE (DEPOSIT(8:), DEP_CURBAL)
      CHARACTER*6     DEP_AMT
      EQUIVALENCE (DEPOSIT(14:), DEP_AMT)
      CHARACTER*6     DEP_NEWBAL
      EQUIVALENCE (DEPOSIT(20:), DEP_NEWBAL)
      CHARACTER*35    DEP_MEMO
      EQUIVALENCE (DEPOSIT(26:), DEP_MEMO)


      INTEGER DEP_AMT_VALUE
      CHARACTER*6     BALANCE_STRING
      CHARACTER*80    DONE

      CALL FDV$CDISP( 'DEPOSIT' )
      CALL CHECK_FMSSTATUS()

      WRITE (BALANCE_STRING, '(I6)') BALANCE

      CALL FDV$PUT( BALANCE_STRING, 'CURBAL' )

C      Get deposit amount and memo from operator.
C      Abort on Kpd Period.
```

```
      CALL FDV$GETAL( DEPOSIT, TERMINATOR )
      IF (TERMINATOR .EQ. FDV$K_KP_PER) RETURN

C      Have deposit information now. If no room in check register
C      must abort.

      IF (LASTREGNUM .EQ. REGSIZE) THEN
          CALL FDV$PUTL( 'Register full, can''t enter deposit' )
          CALL FDV$WAIT
          RETURN
      ENDIF

C      Add to balance and session sum.
C      Check for overflow (program and form keep only six digits).
C      Display new balance.
C      Make entry in register.

      READ (DEP_AMT, '(I6)') DEP_AMT_VALUE
      BALANCE = BALANCE + DEP_AMT_VALUE
      TOTDEP  = TOTDEP  + DEP_AMT_VALUE
      IF (BALANCE .GT. 999999) THEN
          BALANCE = BALANCE - 1000000
          CALL FDV$PUTL( 'Overflow in bank computer, only 6 digits '
     1                   //'allowed, we keep the rest of the money')
          CALL FDV$WAIT
      ENDIF

      WRITE(RI_BALANCE, '(I6)') BALANCE
      CALL FDV$PUT( RI_BALANCE, 'NEWBAL' )
      RI_NUM  = ' '                              ! Blank since it's not a check
      RI_DATE = DEP_DATE
      RI_MEMPAYTO = DEP_MEMO
      RI_AMTDEP = DEP_AMT
      RI_AMTPAY = ' '
      LASTREGNUM = LASTREGNUM + 1
      REGARRAY( LASTREGNUM ) = REGITEM

C      Sample of how to keep message texts stored with the form rather
C      than in a program.  This is especially useful for multi-lingual
C      environments: only the form text and the form named data must
C      be changed and nothing in the program.  The trick is to store the
```

```
C        response text in named data. This is the only example of how to do
C        it in this program, but all messages could be stored like this.
C        Message intent is: "Deposit made, press RETURN or ENTER to continue."

         CALL FDV$RETDN( 'DONE', DONE )
         CALL FDV$PUTL( DONE )
         CALL FDV$WAIT
         RETURN
         END



         SUBROUTINE VIEW_REGISTER

C                View the check register and scroll through it.
C                Also display totals for current session.
C
C        Put up register form.
C        Check for current session totals overflow. If so, output 'OVRFLO'
C        Put out summary of this session into indexed(4) fields.

         IMPLICIT NONE

         INCLUDE 'FDVDEF'
         INCLUDE 'SMPFORTXT(REGISTER_COMMON)'
         INCLUDE 'SMPFORTXT(WORK_AREA)'

         CHARACTER*6 DEPDSP, PAYDSP, BALANCE_STRING
         CHARACTER*2 NSCROL
         INTEGER NSCROL_VALUE

         CHARACTER*100 FAKE

         CALL FDV$CDISP( 'REGISTER' )
         CALL CHECK_FMSSTATUS()

         IF (TOTDEP .LT. 1000000) THEN
             WRITE(DEPDSP, '(I6)') TOTDEP
         ELSE
             DEPDSP = 'OVRFLO'
         ENDIF
```

```
      IF (TOTPAY .LT. 1000000) THEN
          WRITE(PAYDSP, '(I6)') TOTPAY
      ELSE
          PAYDSP = 'OVRFLO'
      ENDIF

      WRITE(BALANCE_STRING, '(I6)') SBALANCE
      CALL FDV$PUT( BALANCE_STRING,    'SUMARY', 1 )
      CALL FDV$PUT( DEPDSP,            'SUMARY', 2 )
      CALL FDV$PUT( PAYDSP,            'SUMARY', 3 )
      WRITE(BALANCE_STRING, '(I6)') BALANCE
      CALL FDV$PUT( BALANCE_STRING,    'SUMARY', 4 )

C      Get number of lines in scroll area from form named data (item 1).

      CALL FDV$RETDI( 1, NSCROL )
      CALL CHECK_FMSSTATUS()

      READ (NSCROL, '(I2)') NSCROL_VALUE

C      Put lines from check resister array into scrolled area.
C      The window is initially from item 1 up to item
C      min(NSCROL,LASTREGNUM), that is, up to the size of the scrolled
C      area or the size of the resister, whichever is less. Assume there
C      is at least one line (the initial deposit).

      MINWINDOW = 1
      CALL FDV$PUTSC( 'NUMBER', REGARRAY(1) )          ! First line
      CURLINE = 1                                      !  Res item cursor is on

      DO WHILE (CURLINE .LT. LASTREGNUM .AND. CURLINE .LT. NSCROL_VALUE)
          CURLINE = CURLINE + 1
          CALL FDV$PFT( FDV$K_FT_SFW, 'NUMBER' )
          CALL FDV$PUTSC( 'NUMBER', REGARRAY( CURLINE ) )
      ENDDO
      MAXWINDOW = CURLINE
```

```
C        Get input from fake field of scrolled line and do what it says:
C               Kpd . or RETURN/ENTER   => return to menu
C               UPARROW or TAB          => scroll forward
C               DOWNARROW or BACKSPACE  => scroll backward
C               all others              => ignore
C        Note that there is no Form Function Key UAR so this routine
C        handles all terminators itself (by ignoring illegal ones).

      CALL FDV$GET( FAKE, TERMINATOR, 'FAKE' )

      DO WHILE (TERMINATOR .NE. FDV$K_FT_NTR .AND.
     1          TERMINATOR .NE. FDV$K_KP_PER )

            IF (TERMINATOR .EQ. FDV$K_FT_SFW .OR.
     1          TERMINATOR .EQ. FDV$K_FT_SNX)  THEN
                  CALL SCROLL_FORWARD()
            ELSE IF (TERMINATOR .EQ. FDV$K_FT_SBK .OR.
     1                TERMINATOR .EQ. FDV$K_FT_SPR)  THEN
                  CALL SCROLL_BACKWARD()
            ENDIF
        CALL FDV$GET( FAKE, TERMINATOR, 'FAKE' )
      ENDDO
      RETURN
      END
```

```
      SUBROUTINE SCROLL_FORWARD

C               CURLINE is the line in the register that the cursor is on.
C               MINWINDOW and MAXWINDOW delimit the part of the register
C               currently displayed in the scrolled area

      IMPLICIT NONE
      INCLUDE 'FDVDEF'
      INCLUDE 'SMPFORTXT(REGISTER_COMMON)'

C      If cursor is at the end of the register, report, and return

      IF (CURLINE .EQ. LASTREGNUM) THEN
          CALL FDV$PUTL( 'Last line of register' )
          RETURN
      ENDIF

C      If cursor not at the last line of a window, just move down
C      If cursor is  at the last line of a window,
C              move window forward one line,
C              write the new last line to the last line of the scrolled area
C      Move current line pointer forward

      IF (CURLINE .NE. 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 ) )
      ENDIF
      CURLINE = CURLINE + 1
      RETURN
      END
```

```
      SUBROUTINE SCROLL_BACKWARD

C               CURLINE is the line in the register that the cursor is on.
CC              MINWINDOW and MAXWINDOW delimit the part of the register
C               currently displayed in the scrolled area

      IMPLICIT NONE
      INCLUDE 'FDVDEF'
      INCLUDE 'SMPFORTXT(REGISTER_COMMON)'

C      If the cursor is at the beginning of the register, report, and return

      IF (CURLINE .EQ. 1) THEN
          CALL FDV$PUTL( 'First line of register' )
          RETURN
      ENDIF

C      If cursor not at first line of the window, just move up
CC     If cursor is  at first line of the window,
CC           move window back one line,
CC           write the new first line to the first line of the scrolled area
C      Move current line pointer back

      IF (CURLINE .NE. 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 ) )
      ENDIF
      CURLINE = CURLINE - 1
      RETURN
      END
```

```
      SUBROUTINE VIEW_ACCOUNT_DATA

C               View the account data.
C               If operator knows the secret word, let operator change
C               the account data for this session.

      IMPLICIT NONE
      INCLUDE 'FDVDEF'
      INCLUDE 'SMPFORTXT(ACCOUNT_COMMON)'
      INCLUDE 'SMPFORTXT(WORK_AREA)'

      CHARACTER*12 PASSWORD

      CALL FDV$CDISP( 'ACCOUNT_DATA' )
      CALL CHECK_FMSSTATUS()

      CALL FDV$PUTAL( ACCOUNT )
      CALL FDV$PUTD( 'SECRET')

C      This is not the best way to do protection, just a way of showing
C      another FMS feature. At this point, supervisor mode is on, so the
C      only input allowed is to the password field.
C      If operator doesn't know password, return to menu.

      CALL FDV$GETAL( , TERMINATOR )          ! Don't care about value now
      IF (TERMINATOR .EQ. FDV$K_KP_PER) RETURN
      CALL FDV$RET( PASSWORD, 'SECRET' )
      IF (OPW .NE. PASSWORD) RETURN

C      Allow input from other fields and read from them.
C      If read is terminated by Keypad period, don't change account.

      CALL FDV$SPOFF
      CALL READ_ALL_FIELDS()
      CALL FDV$SPON                       ! Not really needed, just showing off.
      IF (TERMINATOR .NE. FDV$K_KP_PER) THEN
          CALL FDV$RETAL( ACCOUNT )
          CALL FORMAT_CHECK()
      ENDIF

      RETURN
      END
```

```
      SUBROUTINE READ_ALL_FIELDS

C      Simulate action of FDV$GETAL, using FDV$GETAF and PFT.  Could
C      replace this whole routine with a call on FDV$GETAL, but this shows
C      how mainline program can allow same operator freedom of filling in
C      fields but still regain control after each or changed field.
C      Technique is to read any field, looking only at terminator, then do
C      a process field terminator call to do the operator's action.
C      This technique can be used with calls on FDV$GET or FDV$GETAF.
C      This example starts with a GET on field '*', first field on form.

      IMPLICIT NONE
      INCLUDE 'FDVDEF'
      INCLUDE 'SMPFORTXT(WORK_AREA)'
      INCLUDE 'SMPFORTXT(STATUS_AREA)'

      CHARACTER*6     FIELDNAME
      INTEGER         FIELDINDEX

      CHARACTER*100   JUNK

      CALL FDV$GET( JUNK, TERMINATOR, '*' )
      CALL FDV$RETFN( FIELDNAME, FIELDINDEX )          ! Get first field's name

      DO WHILE (.TRUE.)

          ! Do any special processing for field FIELDNAME at this point.
          ! ...
          ! Go to next or previous field or leave form

          CALL FDV$PFT( TERMINATOR )

          ! If status is error, then PFT failed because terminator was
          ! a keypad key, which means return to caller.

          IF (FMSSTATUS .LT. 0) RETURN

          IF (TERMINATOR .EQ. FDV$K_FT_NTR) THEN
              IF (FMSSTATUS .NE. 2) THEN
                      RETURN
```

```
        ELSE
                CALL FDV$PUTL( 'INPUT REQUIRED' )
                CALL FDV$BELL
        ENDIF
    ENDIF

    ! Go set any other field, returning its name

    CALL FDV$GETAF( JUNK, TERMINATOR, FIELDNAME, FIELDINDEX )
ENDDO
RETURN
END
```

```
      SUBROUTINE GET_AND_CHECK_FMSSTATUS

C      Get the FMS status by calling FDV$STAT.
C      call the routine that checks the status

      IMPLICIT NONE
      INCLUDE 'SMPFORTXT(STATUS_AREA)'

      CALL FDV$STAT( FMSSTATUS, RMSSTATUS )

      CALL CHECK_FMSSTATUS()
      RETURN
      END




      SUBROUTINE CHECK_FMSSTATUS

C      Check FMS status by looking at the status recording variables.

      INCLUDE 'SMPFORTXT(STATUS_AREA)'

      IF (FMSSTATUS .LE. 0) THEN
          !
          ! There is an error returned in the status variables. Detach the
          ! terminal to clean up, then print the errors, and STOP.
          !
          CALL FDV$DTERM( %DESCR(TCA))
          PRINT *,'FDV ERROR.'
          PRINT *,'FMS STATUS:',FMSSTATUS
          IF (FMSSTATUS .EQ. FDV$K_IOL                    .OR.
     1        FMSSTATUS .EQ. FDV$K_IOR) PRINT *,'RMS STATUS:',RMSSTATUS
          STOP
          ENDIF

      RETURN
      END
```

```
      INTEGER FUNCTION VALID1

C     UAR for field validation of any one character field. The
C     UAR associated data has in it the legal characters allowed,
C     except that blank is not allowed unless it appears before
C     the first trailing blank.  For example an assoc. value string
C     'aqr' implies that only the letters a, q, and r are allowed.
C     A string ' aqr' means that blank is acceptable in addition
C     to a, q, and r. Note that this routine is case sensitive
C     (that is, it checks for correct case).  You can get around
C     case sensitivity by using the force upper case field attribute
C     and putting only capitals into the UAR associated value
C     strings.
C
C     This routine can be used with any form and field since
C     it determines the context for itself.

      IMPLICIT NONE

      INCLUDE 'FDVDEF'
      INCLUDE 'SMPFORTXT(WORK_AREA)'

      CHARACTER*31    FRMNAM, FLDNAME
      CHARACTER*80    UARVAL
      CHARACTER*1     FVALUE
      INTEGER         CURPOS, FLDTRM, INSOVR, FINDEX, HELPNUM

C      Retrieve context: we will ignore TCA address, WKSP address, FRMNAM,
C             CURPOS, FLDTRM, and INSOVR, using only UARVAL, and only the
C             initial, non-blank characters of it.
C      Retrieve field name and index.
C      Retrieve field value.

      CALL FDV$RETCX(%DESCR(TCA),%DESCR(WORKSPACE),
     1               FRMNAM, UARVAL, CURPOS, FLDTRM, INSOVR, HELPNUM )
      CALL GET_AND_CHECK_FMSSTATUS
      CALL FDV$RETFN( FLDNAME, FINDEX )
      CALL GET_AND_CHECK_FMSSTATUS
      CALL FDV$RET( FVALUE,  FLDNAME, FINDEX )
      CALL GET_AND_CHECK_FMSSTATUS
```

```
C
C        To be valid, FVALUE must occur in the string UARVAL
C
        IF ( INDEX(UARVAL, FVALUE) .GT. 0) THEN
                VALID1 = FDV$K_UVAL_SUC                 ! Success
        ELSE
                CALL FDV$PUTL( 'Illegal value' )
                CALL GET_AND_CHECK_FMSSTATUS
                VALID1 = FDV$K_UVAL_FAIL
        ENDIF
        RETURN
        END



        INTEGER FUNCTION TAKE15

C        Function Key User Action Routine for the MENU form of SAMP.
C        Convert Keypad 1-5 into field values 1-5.
C        Convert Keypad period into field value 1.
C        Reject all other function keys with error message.

        IMPLICIT NONE

        INCLUDE 'FDVDEF'
        INCLUDE 'SMPFORTXT(WORK_AREA)'

        CHARACTER*4     FRMNAM
        CHARACTER*1     UARVAL, VALUE
        INTEGER         CURPOS, FLDTRM, INSOVR, HELPNUM

C        Retrieve context: we will ignore TCA address, WKSP address, FRMNAM,
C                UARVAL, CURPOS and INSOVR, using only FLDTRM
        CALL FDV$RETCX( %DESCR(TCA), %DESCR(WORKSPACE),
        1               FRMNAM, UARVAL, CURPOS, FLDTRM, INSOVR, HELPNUM)


C        Do the conversion, displaying the value converted if found.
C        Reject if not one of the expected terminators.

        IF (FLDTRM .EQ. FDV$K_KP_1)     THEN
                                                VALUE = '1'
```

```
            ELSE IF (FLDTRM .EQ. FDV$K_KP_2) THEN
                                                    VALUE = '2'
            ELSE IF (FLDTRM .EQ. FDV$K_KP_3) THEN
                                                    VALUE = '3'
            ELSE IF (FLDTRM .EQ. FDV$K_KP_4) THEN
                                                    VALUE = '4'
            ELSE IF (FLDTRM .EQ. FDV$K_KP_5) THEN
                                                    VALUE = '5'
            ELSE IF (FLDTRM .EQ. FDV$K_KP_PER) THEN
                                                    VALUE = '1'

            ELSE
                CALL FDV$PUTL( 'Illegal function key' )
                CALL FDV$SIGOP
                ! Just ignore it now
                TAKE15 = FDV$K_UKEY_SUC
                RETURN
            ENDIF

C           VALUE was legal
C
            CALL FDV$PUT( VALUE, 'OPTION' )
            ! Treat as if it is RETURN
            TAKE15 = FDV$K_UKEY_NTR
            RETURN
            END



            INTEGER FUNCTION PASSKY

C            General function key var to pass only those from the (small) list
C            in the var associated value string and reject all others.
C            The list is of the form: n <oneblank> n <oneblank> ... n <manyblanks>
C            For example the string '110 112' would accept keypad period and
C            keypad zero but no other function keys.

            IMPLICIT NONE

            INCLUDE 'FDVDEF'
            INCLUDE 'SMPFORTXT(WORK_AREA)'
```

```
      CHARACTER*4     FRMNAM
      CHARACTER*82    UARVAL
      INTEGER         CURPOS, FLDTRM, INSOVR, HELPNUM
      INTEGER         NONBLANK, NEXTBLANK, NUMBER_ENTERED

C      Retrieve context: we will ignore TCA address, WKSP address, FRMNAM,
C             INSOVR, and CURPOS, using only FLDTRM and UARVAL.
      CALL FDV$RETCX( %DESCR(TCA), %DESCR(WORKSPACE),
     1                FRMNAM, UARVAL, CURPOS, FLDTRM, INSOVR, HELPNUM )

C      Break up the list into numbers.  Check each against the actual
C      terminator. If terminator found in list, return success.

      NONBLANK = 1              !  Beginning of string

      DO WHILE (UARVAL(NONBLANK:NONBLANK) .NE. ' ')
          NEXTBLANK = INDEX( UARVAL(NONBLANK:), ' ') + NONBLANK - 1
          READ (UARVAL(NONBLANK:), 10) NUMBER_ENTERED
10        FORMAT(I<NEXTBLANK-NONBLANK>)

          IF (FLDTRM .EQ. NUMBER_ENTERED) THEN
              PASSKY = FDV$K_UKEY_TRM         ! Pass key to application
              RETURN
          ENDIF
          NONBLANK = NEXTBLANK + 1
      ENDDO
      PASSKY = FDV$K_UKEY_ERR                 ! Let FDV do the beeping
      RETURN
      END



      INTEGER FUNCTION CHKCHK

C      UAR for SAMP CHECK form.  Makes sure that the check amount is
C      less than or equal to the current balance.  If not, complain and
C      change video attributes on balance field so the potential bouncer
C      can see what there is to work with.

      IMPLICIT NONE
```

```
        INCLUDE 'FDVDEF'

        CHARACTER*6     BALANCE, AMTPAY
        INTEGER         BLINKBOLD, BALANCE_VALUE, AMTPAY_VALUE

        CALL FDV$RET( BALANCE, 'BALANCE' )
        CALL FDV$RET( AMTPAY, 'AMTPAY' )
        READ (BALANCE, '(IG)') BALANCE_VALUE
        READ (AMTPAY,  '(IG)') AMTPAY_VALUE

        IF (BALANCE_VALUE .GE. AMTPAY_VALUE) THEN
            CHKCHK = FDV$K_UVAL_SUC
            BLINKBOLD = -1                          ! Restore to original
            CALL FDV$AFVA( BLINKBOLD, 'BALANCE' )
        ELSE
            CHKCHK = FDV$K_UVAL_FAIL
            BLINKBOLD = 3                           ! Make it very visible
            CALL FDV$AFVA( BLINKBOLD, 'BALANCE' )
            CALL FDV$PUTL( 'Your balance doesn''t cover that much, '
        1                   // 'reenter amount' )
        ENDIF
        RETURN
        END



        INTEGER FUNCTION RANGE

C       General purpose UAR to check the range of any numeric item.  The
C       associated UAR data must have one of the four forms:
C              L,U<space>{message}
C              ,U<space>{message}
C              L,<space>{message}
C              ,<space>{message}
C       where L is lower bound, U is upper bound, and {message} is an
C       optional error message in case the field value is out of bounds.
C       If one of the bounds isn't given, it isn't checked for.  If neither
C       bound is given, nothing is checked, everything succeeds.  If the
C       UAR value doesn't have a comma, a FDV$_UAR error message is returned
C       to the calling program by the FDV so the form designer has to do
```

```
C        back and do it right. If no {message} is given, a simple
C        "out of range U:L" message is given to the hapless operator.
C
C        This UAR can work with any form and numeric field since it gets
C        context itself.  Care must be taken with fields using field marker
C        periods since those periods are not returned to the program.
C
         IMPLICIT NONE

         INCLUDE 'FDVDEF'
         INCLUDE 'SMPFORTXT(WORK_AREA)'

         CHARACTER*31    FRMNAM, NAME
         CHARACTER*80    UARVAL
         CHARACTER*132   NUMBER
         INTEGER         CURPOS, FLDTRM, INSOVR, INDEX_VAL, HELPNUM,
         1               COMMA,  BLANK,  NUMBER_VALUE,   TMP_VALUE

C        Get context which yields associated data value (ignore other stuff).
C        Get current field name and index.
C        Get field value.

         CALL FDV$RETCX( %DESCR(TCA),%DESCR(WORKSPACE),
         1               FRMNAM, UARVAL, CURPOS, FLDTRM, INSOVR, HELPNUM )
         CALL FDV$RETFN( NAME, INDEX_VAL )
         CALL FDV$RET( NUMBER, NAME, INDEX_VAL )
         READ (NUMBER, '(IG)') NUMBER_VALUE
C
C        Find comma and blank delimiters.
C        Check for lower bound.
C
         COMMA = INDEX(UARVAL, ',')
         BLANK = INDEX(UARVAL(COMMA+1:), ' ') + COMMA

         IF (COMMA .EQ. 0) THEN
             RANGE = 0           ! Illegal UARVAL string, FDV returns error
             RETURN
         ENDIF
```

```
        IF (COMMA.NE. 1) THEN
                READ (UARVAL, 10) TMP_VALUE
10              FORMAT (I <COMMA-1>)
                IF (NUMBER_VALUE .LT. TMP_VALUE) GOTO 200
        ENDIF

C       Check for upper bound

        IF (BLANK .NE. COMMA + 1) THEN
                READ (UARVAL(COMMA+1:), 20) TMP_VALUE
20              FORMAT( I <BLANK-COMMA-1>)
                IF (NUMBER_VALUE .GT. TMP_VALUE) GOTO 200
        ENDIF

C        Passed both tests successfully, return success for UAR value

        RANGE = FDV$K_UVAL_SUC
        RETURN

200     CONTINUE

C        Error in one of the bounds.
C        Give error message: either from the UARVAL or make one up.

        IF (UARVAL(BLANK+1:BLANK+1) .NE. ' ') THEN
                CALL FDV$PUTL( UARVAL(BLANK+1:80) )
        ELSE
                CALL FDV$PUTL( 'Field value out of bounds.  Must be in'
        1               // ' range "' // UARVAL(1:BLANK-1) // '".')
        ENDIF

        CALL FDV$SIGOP                  ! Beep, too.
        RANGE = FDV$K_UVAL_FAIL
        RETURN
        END
```

```
!*****************************************************************************
! FDVDEF.FOR  --  This is the include file for FMS applications in FORTRAN *
!*****************************************************************************

!*************************
! FMS terminator codes: *
!*************************
        INTEGER     FDV$K_FT_NTR
        PARAMETER ( FDV$K_FT_NTR      =  0 ) !Enter (i.e. end GETs)
        INTEGER     FDV$K_FT_NXT
        PARAMETER ( FDV$K_FT_NXT      =  1 ) !Next field
        INTEGER     FDV$K_FT_PRV
        PARAMETER ( FDV$K_FT_PRV      =  2 ) !Previous field
        INTEGER     FDV$K_FT_ATB
        PARAMETER ( FDV$K_FT_ATB      =  3 ) !Automatically move to next field
        INTEGER     FDV$K_FT_XBK
        PARAMETER ( FDV$K_FT_XBK      =  4 ) !Exit scrolled area backward
        INTEGER     FDV$K_FT_XFW
        PARAMETER ( FDV$K_FT_XFW      =  5 ) !Exit scrolled area forward
        INTEGER     FDV$K_FT_SNX
        PARAMETER ( FDV$K_FT_SNX      =  6 ) !Scroll forward to next field
        INTEGER     FDV$K_FT_SPR
        PARAMETER ( FDV$K_FT_SPR      =  7 ) !Scroll backward to previous field
        INTEGER     FDV$K_FT_SFW
        PARAMETER ( FDV$K_FT_SFW      =  8 ) !Scroll forward
        INTEGER     FDV$K_FT_SBK
        PARAMETER ( FDV$K_FT_SBK      =  9 ) !Scroll backward
        INTEGER     FDV$K_FT_ILG_NXT
        PARAMETER ( FDV$K_FT_ILG_NXT = 11 ) !Ils ctxt: next field
        INTEGER     FDV$K_FT_ILG_PRV
        PARAMETER ( FDV$K_FT_ILG_PRV = 12 ) !Ils ctxt: previous field
        INTEGER     FDV$K_FT_ILG_ATB
        PARAMETER ( FDV$K_FT_ILG_ATB = 13 ) !Ils ctxt: auto move to next field
        INTEGER     FDV$K_FT_ILG_XBK
        PARAMETER ( FDV$K_FT_ILG_XBK = 14 ) !Ils ctxt: exit scr area backward
        INTEGER     FDV$K_FT_ILG_XFW
        PARAMETER ( FDV$K_FT_ILG_XFW = 15 ) !Ils ctxt: exit scr area forward
        INTEGER     FDV$K_FT_ILG_SFW
        PARAMETER ( FDV$K_FT_ILG_SFW = 16 ) !Ils ctxt: scroll forward
        INTEGER     FDV$K_FT_ILG_SBK
        PARAMETER ( FDV$K_FT_ILG_SBK = 17 ) !Ils ctxt: scroll backward
```

```
!***********************************************************
! Function Key terminators returned from GETs and WAIT *
! Also used as FDV Keycodes for use with DFKBD.        *
!***********************************************************
        INTEGER     FDV$K_AR_UP
        PARAMETER ( FDV$K_AR_UP      =   99 )
        INTEGER     FDV$K_AR_DOWN
        PARAMETER ( FDV$K_AR_DOWN    =  100 )
        INTEGER     FDV$K_AR_RIGHT
        PARAMETER ( FDV$K_AR_RIGHT   =  101 )
        INTEGER     FDV$K_AR_LEFT
        PARAMETER ( FDV$K_AR_LEFT    =  102 )
        INTEGER     FDV$K_PF_1
        PARAMETER ( FDV$K_PF_1       =  103 )
        INTEGER     FDV$K_PF_2
        PARAMETER ( FDV$K_PF_2       =  104 )
        INTEGER     FDV$K_PF_3
        PARAMETER ( FDV$K_PF_3       =  105 )
        INTEGER     FDV$K_PF_4
        PARAMETER ( FDV$K_PF_4       =  106 )
        INTEGER     FDV$K_KP_NTR
        PARAMETER ( FDV$K_KP_NTR     =  107 )
        INTEGER     FDV$K_KP_COM
        PARAMETER ( FDV$K_KP_COM     =  108 )
        INTEGER     FDV$K_KP_HYP
        PARAMETER ( FDV$K_KP_HYP     =  109 )
        INTEGER     FDV$K_KP_PER
        PARAMETER ( FDV$K_KP_PER     =  110 )
        INTEGER     FDV$K_KP_0
        PARAMETER ( FDV$K_KP_0       =  112 )
        INTEGER     FDV$K_KP_1
        PARAMETER ( FDV$K_KP_1       =  113 )
        INTEGER     FDV$K_KP_2
        PARAMETER ( FDV$K_KP_2       =  114 )
        INTEGER     FDV$K_KP_3
        PARAMETER ( FDV$K_KP_3       =  115 )
        INTEGER     FDV$K_KP_4
        PARAMETER ( FDV$K_KP_4       =  116 )
        INTEGER     FDV$K_KP_5
        PARAMETER ( FDV$K_KP_5       =  117 )
        INTEGER     FDV$K_KP_6
        PARAMETER ( FDV$K_KP_6       =  118 )
```

```
INTEGER      FDV$K_KP_7
PARAMETER ( FDV$K_KP_7       =  119 )
INTEGER      FDV$K_KP_8
PARAMETER ( FDV$K_KP_8       =  120 )
INTEGER      FDV$K_KP_9
PARAMETER ( FDV$K_KP_9       =  121 )
INTEGER      FDV$K_GAR_UP
PARAMETER ( FDV$K_GAR_UP     =  227 )
INTEGER      FDV$K_GAR_DOWN
PARAMETER ( FDV$K_GAR_DOWN   =  228 )
INTEGER      FDV$K_GAR_RIGHT
PARAMETER ( FDV$K_GAR_RIGHT  =  229 )
INTEGER      FDV$K_GAR_LEFT
PARAMETER ( FDV$K_GAR_LEFT   =  230 )
INTEGER      FDV$K_GPF_1
PARAMETER ( FDV$K_GPF_1      =  231 )
INTEGER      FDV$K_GPF_2
PARAMETER ( FDV$K_GPF_2      =  232 )
INTEGER      FDV$K_GPF_3
PARAMETER ( FDV$K_GPF_3      =  233 )
INTEGER      FDV$K_GPF_4
PARAMETER ( FDV$K_GPF_4      =  234 )
INTEGER      FDV$K_GKP_NTR
PARAMETER ( FDV$K_GKP_NTR    =  235 )
INTEGER      FDV$K_GKP_COM
PARAMETER ( FDV$K_GKP_COM    =  236 )
INTEGER      FDV$K_GKP_HYP
PARAMETER ( FDV$K_GKP_HYP    =  237 )
INTEGER      FDV$K_GKP_PER
PARAMETER ( FDV$K_GKP_PER    =  238 )
INTEGER      FDV$K_GKP_0
PARAMETER ( FDV$K_GKP_0      =  240 )
INTEGER      FDV$K_GKP_1
PARAMETER ( FDV$K_GKP_1      =  241 )
INTEGER      FDV$K_GKP_2
PARAMETER ( FDV$K_GKP_2      =  242 )
INTEGER      FDV$K_GKP_3
PARAMETER ( FDV$K_GKP_3      =  243 )
INTEGER      FDV$K_GKP_4
PARAMETER ( FDV$K_GKP_4      =  244 )
INTEGER      FDV$K_GKP_5
PARAMETER ( FDV$K_GKP_5      =  245 )
```

```
        INTEGER     FDV$K_GKP_6
        PARAMETER ( FDV$K_GKP_6     =  246 )
        INTEGER     FDV$K_GKP_7
        PARAMETER ( FDV$K_GKP_7     =  247 )
        INTEGER     FDV$K_GKP_8
        PARAMETER ( FDV$K_GKP_8     =  248 )
        INTEGER     FDV$K_GKP_9
        PARAMETER ( FDV$K_GKP_9     =  249 )
!*****************************************
! FDV Keyfunctions. For use in DFKBD call. *
!*****************************************
        INTEGER     FDV$K_KF_GOLD
        PARAMETER ( FDV$K_KF_GOLD   =    1 )
        INTEGER     FDV$K_KF_RESET
        PARAMETER ( FDV$K_KF_RESET  =    2 )
        INTEGER     FDV$K_KF_CRSLF
        PARAMETER ( FDV$K_KF_CRSLF  =    3 )
        INTEGER     FDV$K_KF_CRSRT
        PARAMETER ( FDV$K_KF_CRSRT  =    4 )
        INTEGER     FDV$K_KF_DLCHR
        PARAMETER ( FDV$K_KF_DLCHR  =    5 )
        INTEGER     FDV$K_KF_DLFLD
        PARAMETER ( FDV$K_KF_DLFLD  =    6 )
        INTEGER     FDV$K_KF_INS
        PARAMETER ( FDV$K_KF_INS    =    7 )
        INTEGER     FDV$K_KF_OVR
        PARAMETER ( FDV$K_KF_OVR    =    8 )
        INTEGER     FDV$K_KF_RFRSH
        PARAMETER ( FDV$K_KF_RFRSH  =    9 )
        INTEGER     FDV$K_KF_HELP
        PARAMETER ( FDV$K_KF_HELP   =   10 )
        INTEGER     FDV$K_KF_NXT
        PARAMETER ( FDV$K_KF_NXT    =   11 )
        INTEGER     FDV$K_KF_PRV
        PARAMETER ( FDV$K_KF_PRV    =   12 )
        INTEGER     FDV$K_KF_NTR
        PARAMETER ( FDV$K_KF_NTR    =   13 )
        INTEGER     FDV$K_KF_SBK
        PARAMETER ( FDV$K_KF_SBK    =   14 )
        INTEGER     FDV$K_KF_SFW
        PARAMETER ( FDV$K_KF_SFW    =   15 )
        INTEGER     FDV$K_KF_XBK
```

```
        PARAMETER ( FDV$K_KF_XBK    =  16 )
        INTEGER     FDV$K_KF_XFW
        PARAMETER ( FDV$K_KF_XFW    =  17 )
        INTEGER     FDV$K_KF_NONE
        PARAMETER ( FDV$K_KF_NONE   =   0 )
        INTEGER     FDV$K_KF_DFLT
        PARAMETER ( FDV$K_KF_DFLT   = -1 )
!***************************************************************
! UAR return codes. These codes are returned by UAR to FDV. *
!***************************************************************
! Field completion return codes *
!*******************************
        INTEGER     FDV$K_UVAL_SUC
        PARAMETER ( FDV$K_UVAL_SUC =  1000 )    !Field completion success
        INTEGER     FDV$K_UVAL_FAIL
        PARAMETER ( FDV$K_UVAL_FAIL = 1001 )    !Field completion failure
        INTEGER     FDV$K_UVAL_END
        PARAMETER ( FDV$K_UVAL_END =  1002 )    !Field completion suc-stop UAR
!*************************
! Help UAR return codes *
!*************************
        INTEGER     FDV$K_UHELP_NO
        PARAMETER ( FDV$K_UHELP_NO =  2000 )    !No help given, try next step
        INTEGER     FDV$K_UHELPED
        PARAMETER ( FDV$K_UHELPED  =  2001 )    !Help given, continue sequence
        INTEGER     FDV$K_UHELP_ALL
        PARAMETER ( FDV$K_UHELP_ALL = 2002 )    !Help given, repeat UAR
!*********************************
! Function Key UAR return codes *
!*********************************
        INTEGER     FDV$K_UKEY_ERR
        PARAMETER ( FDV$K_UKEY_ERR =  3000 )    !Fn Key failure, FDV signals
        INTEGER     FDV$K_UKEY_TRM
        PARAMETER ( FDV$K_UKEY_TRM =  3001 )    !Fn Key success, normal f.K.
        INTEGER     FDV$K_UKEY_NXT
        PARAMETER ( FDV$K_UKEY_NXT =  3002 )    !Fn Key succ, treat as NEXT
        INTEGER     FDV$K_UKEY_NTR
        PARAMETER ( FDV$K_UKEY_NTR =  3003 )    !Fn Key succ, treat as ENTER
        INTEGER     FDV$K_UKEY_SUC
        PARAMETER ( FDV$K_UKEY_SUC =  3004 )    !Fn Key succ, ignore
```

```
!****************************************************************************
! FDV status codes returned when FDV$... routines are called as functions. *
! These codes are VMS status codes and can be signalled.  They correspond  *
! one-to-one with the FMS status codes retrievable from FDV$STAT.          *
!****************************************************************************
        INTEGER FDV$_SUC
        PARAMETER ( FDV$_SUC = 2719889 )
        INTEGER FDV$_INC
        PARAMETER ( FDV$_INC = 2719897 )
        INTEGER FDV$_MOD
        PARAMETER ( FDV$_MOD = 2719905 )
        INTEGER FDV$_IMP
        PARAMETER ( FDV$_IMP = 2719922 )
        INTEGER FDV$_FSP
        PARAMETER ( FDV$_FSP = 2719930 )
        INTEGER FDV$_IOL
        PARAMETER ( FDV$_IOL = 2719938 )
        INTEGER FDV$_FLB
        PARAMETER ( FDV$_FLB = 2719946 )
        INTEGER FDV$_ICH
        PARAMETER ( FDV$_ICH = 2719954 )
        INTEGER FDV$_FCH
        PARAMETER ( FDV$_FCH = 2719962 )
        INTEGER FDV$_FRM
        PARAMETER ( FDV$_FRM = 2719970 )
        INTEGER FDV$_FNM
        PARAMETER ( FDV$_FNM = 2719978 )
        INTEGER FDV$_LIN
        PARAMETER ( FDV$_LIN = 2719986 )
        INTEGER FDV$_FLD
        PARAMETER ( FDV$_FLD = 2719994 )
        INTEGER FDV$_NOF
        PARAMETER ( FDV$_NOF = 2720002 )
        INTEGER FDV$_DSP
        PARAMETER ( FDV$_DSP = 2720010 )
        INTEGER FDV$_NSC
        PARAMETER ( FDV$_NSC = 2720018 )
        INTEGER FDV$_DNM
        PARAMETER ( FDV$_DNM = 2720026 )
        INTEGER FDV$_DLN
        PARAMETER ( FDV$_DLN = 2720034 )
        INTEGER FDV$_UTR
```

```
PARAMETER ( FDV$_UTR = 2720042 )
INTEGER FDV$_IOR
PARAMETER ( FDV$_IOR = 2720050 )
INTEGER FDV$_IFN
PARAMETER ( FDV$_IFN = 2720058 )
INTEGER FDV$_ARG
PARAMETER ( FDV$_ARG = 2720066 )
INTEGER FDV$_INI
PARAMETER ( FDV$_INI = 2720074 )
INTEGER FDV$_STR
PARAMETER ( FDV$_STR = 2720082 )
INTEGER FDV$_IVM
PARAMETER ( FDV$_IVM = 2720090 )
INTEGER FDV$_FVM
PARAMETER ( FDV$_FVM = 2720098 )
INTEGER FDV$_ITT
PARAMETER ( FDV$_ITT = 2720106 )
INTEGER FDV$_TCA
PARAMETER ( FDV$_TCA = 2720114 )
INTEGER FDV$_STA
PARAMETER ( FDV$_STA = 2720122 )
INTEGER FDV$_WID
PARAMETER ( FDV$_WID = 2720130 )
INTEGER FDV$_NFL
PARAMETER ( FDV$_NFL = 2720138 )
INTEGER FDV$_IBF
PARAMETER ( FDV$_IBF = 2720146 )
INTEGER FDV$_NDS
PARAMETER ( FDV$_NDS = 2720154 )
INTEGER FDV$_UDP
PARAMETER ( FDV$_UDP = 2720162 )
INTEGER FDV$_UAR
PARAMETER ( FDV$_UAR = 2720170 )
INTEGER FDV$_UNF
PARAMETER ( FDV$_UNF = 2720178 )
INTEGER FDV$_CAN
PARAMETER ( FDV$_CAN = 2720194 )
INTEGER FDV$_KIF
PARAMETER ( FDV$_KIF = 2720202 )
INTEGER FDV$_KEX
PARAMETER ( FDV$_KEX = 2720210 )
INTEGER FDV$_KTW
```

```
        PARAMETER ( FDV$_KTW = 2720218 )
        INTEGER FDV$_KIL
        PARAMETER ( FDV$_KIL = 2720226 )
        INTEGER FDV$_TMO
        PARAMETER ( FDV$_TMO = 2720234 )
        INTEGER FDV$_LLI
        PARAMETER ( FDV$_LLI = 2720242 )
        INTEGER FDV$_VAL
        PARAMETER ( FDV$_VAL = 2720250 )
        INTEGER FDV$_IFU
        PARAMETER ( FDV$_IFU = 2720258 )
        INTEGER FDV$_SYS
        PARAMETER ( FDV$_SYS = 2720266 )
!****************************************************************************
! FMS status codes returned when FDV$STAT routine is called.                *
!****************************************************************************
! Success codes.
        INTEGER     FDV$K_SUC
        PARAMETER ( FDV$K_SUC =  1 )
        INTEGER     FDV$K_INC
        PARAMETER ( FDV$K_INC =  2 )
        INTEGER     FDV$K_MOD
        PARAMETER ( FDV$K_MOD =  3 )

! Failure codes
        INTEGER     FDV$K_IMP
        PARAMETER ( FDV$K_IMP =  -2 )
        INTEGER     FDV$K_FSP
        PARAMETER ( FDV$K_FSP =  -3 )
        INTEGER     FDV$K_IOL
        PARAMETER ( FDV$K_IOL =  -4 )
        INTEGER     FDV$K_FLB
        PARAMETER ( FDV$K_FLB =  -5 )
        INTEGER     FDV$K_ICH
        PARAMETER ( FDV$K_ICH =  -6 )
        INTEGER     FDV$K_FCH
        PARAMETER ( FDV$K_FCH =  -7 )
        INTEGER     FDV$K_FRM
        PARAMETER ( FDV$K_FRM =  -8 )
        INTEGER     FDV$K_FNM
        PARAMETER ( FDV$K_FNM =  -9 )
        INTEGER     FDV$K_LIN
```

```
PARAMETER ( FDV$K_LIN =  -10 )
INTEGER     FDV$K_FLD
PARAMETER ( FDV$K_FLD =  -11 )
INTEGER     FDV$K_NOF
PARAMETER ( FDV$K_NOF =  -12 )
INTEGER     FDV$K_DSP
PARAMETER ( FDV$K_DSP =  -13 )
INTEGER     FDV$K_NSC
PARAMETER ( FDV$K_NSC =  -14 )
INTEGER     FDV$K_DNM
PARAMETER ( FDV$K_DNM =  -15 )
INTEGER     FDV$K_DLN
PARAMETER ( FDV$K_DLN =  -16 )
INTEGER     FDV$K_UTR
PARAMETER ( FDV$K_UTR =  -17 )
INTEGER     FDV$K_IOR
PARAMETER ( FDV$K_IOR =  -18 )
INTEGER     FDV$K_IFN
PARAMETER ( FDV$K_IFN =  -19 )
INTEGER     FDV$K_ARG
PARAMETER ( FDV$K_ARG =  -20 )
INTEGER     FDV$K_INI
PARAMETER ( FDV$K_INI =  -21 )
INTEGER     FDV$K_STR
PARAMETER ( FDV$K_STR =  -22 )
INTEGER     FDV$K_FVM
PARAMETER ( FDV$K_FVM =  -23 )
INTEGER     FDV$K_IVM
PARAMETER ( FDV$K_IVM =  -24 )
INTEGER     FDV$K_ITT
PARAMETER ( FDV$K_ITT =  -25 )
INTEGER     FDV$K_TCA
PARAMETER ( FDV$K_TCA =  -26 )
INTEGER     FDV$K_STA
PARAMETER ( FDV$K_STA =  -27 )
INTEGER     FDV$K_WID
PARAMETER ( FDV$K_WID =  -28 )
INTEGER     FDV$K_NFL
PARAMETER ( FDV$K_NFL =  -29 )
INTEGER     FDV$K_IBF
PARAMETER ( FDV$K_IBF =  -30 )
INTEGER     FDV$K_NDS
```

```
PARAMETER ( FDV$K_NDS =  -31 )
INTEGER     FDV$K_UDP
PARAMETER ( FDV$K_UDP =  -33 )
INTEGER     FDV$K_UAR
PARAMETER ( FDV$K_UAR =  -34 )
INTEGER     FDV$K_UNF
PARAMETER ( FDV$K_UNF =  -35 )
INTEGER     FDV$K_CAN
PARAMETER ( FDV$K_CAN =  -39 )
INTEGER     FDV$K_KIF
PARAMETER ( FDV$K_KIF =  -40 )
INTEGER     FDV$K_KEX
PARAMETER ( FDV$K_KEX =  -41 )
INTEGER     FDV$K_KTW
PARAMETER ( FDV$K_KTW =  -42 )
INTEGER     FDV$K_KIL
PARAMETER ( FDV$K_KIL =  -43 )
INTEGER     FDV$K_TMO
PARAMETER ( FDV$K_TMO =  -44 )
INTEGER     FDV$K_LLI
PARAMETER ( FDV$K_LLI =  -45 )
INTEGER     FDV$K_VAL
PARAMETER ( FDV$K_VAL =  -47 )
INTEGER     FDV$K_IFU
PARAMETER ( FDV$K_IFU =  -48 )
INTEGER     FDV$K_SYS
PARAMETER ( FDV$K_SYS =  -49 )
!**************************************************
! Declare the FDV routines                        *
!**************************************************
    INTEGER FDV$ADLVA
    INTEGER FDV$AFVA
    INTEGER FDV$ATERM
    INTEGER FDV$AWKSP
    INTEGER FDV$BELL
    INTEGER FDV$CANCL
    INTEGER FDV$CDISP
    INTEGER FDV$CLEAR
    INTEGER FDV$DEL
    INTEGER FDV$DFKBD
    INTEGER FDV$DISP
    INTEGER FDV$DPCOM
```

```
      INTEGER FDV$DTERM
      INTEGER FDV$DWKSP
      INTEGER FDV$GET
      INTEGER FDV$GETAF
      INTEGER FDV$GETAL
      INTEGER FDV$GETDL
      INTEGER FDV$GETSC
      INTEGER FDV$ILTRM
      INTEGER FDV$LCHAN
      INTEGER FDV$LCLOS
      INTEGER FDV$LEDOF
      INTEGER FDV$LEDON
      INTEGER FDV$LOAD
      INTEGER FDV$LOPEN
      INTEGER FDV$NDISP
      INTEGER FDV$PFT
      INTEGER FDV$PUT
      INTEGER FDV$PUTAL
      INTEGER FDV$PUTD
      INTEGER FDV$PUTDA
      INTEGER FDV$PUTL
      INTEGER FDV$PUTSC
      INTEGER FDV$READ
      INTEGER FDV$RET
      INTEGER FDV$RETAL
      INTEGER FDV$RETCX
      INTEGER FDV$RETDI
      INTEGER FDV$RETDN
      INTEGER FDV$RETFL
      INTEGER FDV$RETFN
      INTEGER FDV$RETFO
      INTEGER FDV$RETLE
      INTEGER FDV$RFRSH
      INTEGER FDV$SIGOP
      INTEGER FDV$SPADA
      INTEGER FDV$SPOFF
      INTEGER FDV$SPON
      INTEGER FDV$SSIGQ
      INTEGER FDV$SSRV
      INTEGER FDV$STAT
      INTEGER FDV$SWKSP
      INTEGER FDV$WAIT
```

Chapter 7
Programming FMS Applications in VAX–11 PASCAL

The FMS Form Driver processes all Form Driver calls according to VAX–11 standards and language-specific rules. These rules define how parameters are passed to the Form Driver and how values are returned to your program. Language-specific information is briefly presented in this manual. For more detail, refer to the VAX–11 PASCAL document set.

Your VAX–11 PASCAL application program must comply with the requirements of the VAX–11 PASCAL FMS interface. Topics discussed in this chapter include:

- Form Driver Routines
  - Invoking Form Driver Routines as Procedures
  - Accessing Form Driver Status Codes as Functions
- Parameter Passing in FMS
- Null Arguments
- Entry Point Definitions
- FMS Data Types
  - Character Strings
  - Longword Binary Integers
  - Word Binary Integers
- Non-FMS Data Types
- One-Dimensional Arrays
- Allocation: Workspace, Terminal Control Area, and Run-Time Memory-Resident Form Area
- Precautions for Using FMS
- Data Conversion
- Sample Application Program in VAX–11 PASCAL

A sample program written in PASCAL (SAMPPAS.PAS) appears at the end of this chapter. Following the code for the Sample Application are Form Driver environment files which you may wish to include in your own application program. Command file information needed to build the Sample Application program is in Section 7.11.2.

Examples from the Sample Application are used throughout the text to illustrate language issues. Where appropriate examples from SAMPPAS.PAS do not exist, other examples are provided.

7.1 Form Driver Routines

You can call any FMS routine as a procedure or as a function. Syntax follows standard VAX–11 PASCAL requirements.

7.1.1 Invoking Form Driver Routines as Procedures

You can invoke a Form Driver routine as a procedure as shown in the following examples:

```
FDV$WAIT ();
```

Calls the Form Driver routine FDV$WAIT and passes no parameters.

```
FDV$GET (FLDVAL := Option, FLDTRM := Terminator, FLDNAM := 'OPTION');
```

Calls the Form Driver routine FDV$GET and passes three parameters.

See Appendix A for a complete list of Form Driver calls. The calling sequence for each Form Driver routine, data access codes, data types, and passing mechanisms are presented in language-independent notation specified by the VAX–11 Procedure Calling and Condition Handling Standard. For further detail about the VAX–11 Procedure Calling and Condition Handling Standard, refer to the *VAX–11 Run-Time Library Reference Manual.*

7.1.2 Accessing Form Driver Status Codes as Functions

An FMS status code is returned to the calling program at the completion of all Form Driver calls. To receive the returned status code from a Form Driver routine, you activate the routine with a function designator. Note that this returns a standard VMS status code. For portability, other status mechanisms can also be used. (For more information, see the *VAX–11 FMS Form Driver Reference Manual*, Chapter 2.)

The following statement calls FDV$GET as an FMS function:

```
STATUS_RETURN = FDV$GET (OPTION, TERMINATOR, 'OPTION');
```

7.2 Parameter Passing in FMS

The parameter passing mechanism refers to the way in which data is passed to a called routine. The VAX–11 Procedure Calling Standard has three methods for passing parameters:

- By reference
- By descriptor
- By value

FMS routines, however, expect parameters to be passed only by reference and by descriptor.

By reference specifies that the storage location of the parameter is passed to the routine. FMS expects integers to be passed by reference which is the PASCAL default passing mechanism for all data types.

By descriptor specifies that the address of a descriptor data structure is passed to the called routine. FMS expects character strings and arrays to be passed by descriptor. The default passing mechanism for conformant arrays (those that assume the characteristics of their actual arguments) is by descriptor. However, the PASCAL default passing mechanism for character strings is by reference. Consequently, the [CLASS_S] attribute is used to force use of the [CLASS_S] string descriptor mechanism for passing character strings to the FMS routine. For example:

```
[ASYNCHRONOUS] FUNCTION FDV$RETCX (
   VAR tca : [VOLATILE] ARRAY [$11..$u1:INTEGER] OF INTEGER;
   VAR wksp : [VOLATILE] ARRAY [$12..$u2:INTEGER] OF INTEGER;
   VAR frmnam : [CLASS_S] PACKED ARRAY [$13..$u3:INTEGER] OF CHAR;
   VAR uarval : [CLASS_S] PACKED ARRAY [$14..$u4:INTEGER] OF CHAR;
   VAR curpos : [VOLATILE] INTEGER;
   VAR fldtrm : [VOLATILE] INTEGER;
   VAR insovr : [VOLATILE] INTEGER;
   VAR hlpnum : [VOLATILE] INTEGER) : INTEGER; EXTERNAL;
```

7.3 Null Arguments

When the call syntax includes optional parameters and you do not wish to specify all of the information, you can use null arguments. Each optional parameter can be omitted to simplify your program. Optional parameters to the right of the last required parameter can simply be omitted from the call. In the following example, the FDV$GETAL call passes only the field terminator value:

```
FDV$GETAL (FLDTRM := Terminator);
```

PASCAL has two ways to specify null arguments. One approach is to use nonpositional syntax. You use named parameters and do not reference the

omitted parameters. The other approach is to use positional parameters and represent null parameters by a comma. Thus, the following two statements are equivalent:

```
FDV$GETAL (FLDTRM := Terminator)
FDV$GETAL (Terminator,,,)
```

All optional parameters must be declared with a default value (usually %IMMED 0).

VAX–11 PASCAL supports null arguments by allowing you to supply a default immediate value of zero in the declaration of the routine to which the null argument is to be passed. For example, in the declaration of the function FDV$ATERM, a default immediate value of zero is supplied to the parameters size, channel, and terminal.

```
[ASYNCHRONOUS] FUNCTION FDV$ATERM (
    VAR tca : [VOLATILE] ARRAY [$l1..$u1 : INTEGER] OF INTEGER;
    size : INTEGER := %IMMED 0;
    channel : INTEGER := %IMMED 0;
    terminal : INTEGER := %IMMED 0) : INTEGER; EXTERNAL
```

A call to the FDV$ATERM routine declared above follows:

```
FDV$ATERM (terminal, 12, 1,);
```

The following call is equivalent but it leaves off the trailing comma:

```
FDV$ATERM (terminal, 12, 1);
```

7.4 Entry Point Definitions

The most difficult part of calling external routines from PASCAL is defining the entry points. Every entry point used in a PASCAL program must be declared with all its parameters and their types.

It is extremely important to have complete, correct definitions of all the entry points and their arguments. Your program will not compile if the number, data types, and uses of arguments in a call do not agree with their declarations. The include file FDVDEF.PAS contains definitions for the Form Driver constants and entry points. To access the Form Driver entry points, your program must inherit the precompiled environment file FDVDEF.PEN. The following steps can be performed:

1. Obtain the source file FDVDEF.PAS from the directory called FMS$EXAMPLES.
2. Compile the file FDVDEF.PAS to produce the precompiled environment file FDVDEF.PEN:

   ```
   $ PASCAL/ENVIRONMENT FDVDEF.PAS
   ```

3. Incorporate the precompiled environment file FDVDEF.PEN into your program:

   ```
   [INHERIT('FDVDEF.PEN')]PROGRAM name ...;
   ```

Many calls to FMS have a variable number of arguments. If you use the file FDVDEF.PEN, you do not have to worry about these variations in specified arguments because FDVDEF.PEN has the entry points for the Form Driver defined.

7.5 FMS Data Types

7.5.1 Character Strings

The character string is one of the general data types used by FMS. For example, the FDV$GET call passes the character strings for field value (Option) and field name ('OPTION'):

```
FDV$GET (OPTION, TERMINATOR, 'OPTION');
```

7.5.1.1 Declaring Fixed-Length Strings — Although FMS accepts both varying-length and fixed-length strings as parameters, it treats all strings as if they were fixed length. In other words, FMS does not alter the length of a varying-length string descriptor when it returns values to the output parameters. When you use fixed-length strings, you must be certain that your strings are initially declared to be long enough to accommodate your FMS data. When you use varying-length strings, be certain that the upper boundary of the string is large enough to accommodate the maximum string length expected for that variable.

Two approaches are available for satisfying the fixed-length string constraints of FMS. One option is to declare your fixed-length strings to be the exact length of the FMS data to be returned. You can use the FMS/DESCRIPTION/DECLARATIONS command to get the length of the strings.

Alternatively, a single string variable can be used in different FMS calls to transfer data to or from several forms and fields. You must declare the string variable to be at least as large as the longest field value string that will be returned to your program. You can use the FMS/DESCRIPTION/BRIEF command to get this information. Use the FDV$RETLE call to return the length of the valid portion of the field value in the string variable. This length can then be used when referencing the data that was entered in the field.

```
FDV$GET (ACCOUNT, TERMINATOR, 'FIELD');
FDV$RETLE (LENGTHFIELD, 'FIELD');

WRITE (SUBSTR (ACCOUNT, 1, LENGTHFIELD));
```

After the FDV$RETLE call, LENGTHFIELD is equal to the length of the field named 'FIELD.' It is also equal to the the valid portion of the string that is defined by the string descriptor ACCOUNT. LENGTHFIELD can now be used to reference the data that was entered in the field named 'FIELD'. If you do not use the PASCAL SUBSTR function when designating ACCOUNT, you will designate the entire variable, including any blanks used by the Form Driver to pad the string.

A useful application of the FDV$RETLE call is in general purpose user action routines.

7.5.2 Longword Binary Integers

The longword binary integer is another general data type used by FMS. For example, the FDV$ATERM call passes the longword value for terminal control area size (12) and logical I/O channel number (2):

```
FDV$ATERM (TCA := Tca, Size := 12, Channel := 2);
```

Numeric arguments must be longword binary integers. If you try to pass other numeric types to the Form Driver, the calls do not work properly. An exception is the FDV$DFKBD call (see the next section).

7.5.3 Word Binary Integers

The defkbd argument is a word integer array passed when the FDV$DFKBD routine is called. FMS expects arrays to be passed by descriptor.

7.6 Non-FMS Data Types

PASCAL data types that are not recognized by FMS can be used in your PASCAL application program provided they are not passed to the Form Driver.

7.7 One-Dimensional Arrays

One-dimensional arrays are structures that can be used in FMS for the following arguments:

- tca (terminal control area)
- wksp (workspace)
- mloc (memory location)
- defkbd (define keyboard)

You must provide FMS with storage space for these arguments. You can do that by declaring them to be:

- longword integer arrays or character strings for tca, wksp, and mloc
- word integer arrays for defkbd

In the Sample Application program, the tca, wksp, and mloc arguments are passed to several Form Driver routines. These arguments are declared to be integer array variables. You may alternatively declare these variables to be character strings. (The strings can be static or varying length but must be extended to the proper length.) If you declare these variables to be char-

acter strings, you need to redefine all of the entry points that reference terminal control area, workspace, and memory location. Otherwise, you will get compile errors.

The following declarations establish names and storage for the integer array variables Workspace, Checkwksp, Tca, and Menu_form:

```
VAR  Workspace : [VOLATILE] ARRAY [1..3] OF INTEGER; { General workspace }
     Checkwksp : [VOLATILE] ARRAY [1..3] OF INTEGER; { Check workspace }
     Tca : [VOLATILE] ARRAY [1..3] OF INTEGER;       { Term Control Area }
     Menu_form : [VOLATILE] ARRAY [1..500] OF INTEGER;
                                  { Storage for memory-resident forms }
```

7.8 Allocation: Workspace, Terminal Control Area, and Run-Time Memory-Resident Form Area

The FMS workspace, terminal control area, and run-time memory-resident form area variables are character strings or longword integer arrays. These variables should be declared with the VOLATILE attribute.

The Form Driver needs 12 bytes to associate user information about each workspace and terminal control area. The allocation for a run-time memory-resident form area must be the size of the form. The space for all of these variables is allocated by your application program. Note that FMS uses *only* 12 bytes of space for the workspace and terminal control area allocation. Any allocation greater than 12 bytes is wasted. You declare your workspace, terminal control area, and run-time memory-resident form area only once. FMS remembers their addresses after the addresses have been initially passed to the Form Driver.

For each workspace, the Form Driver also allocates an additional amount of storage space based on your estimate of the amount of memory needed to store your largest form. If your estimate is too small, the Form Driver will allocate more space automatically, but performance may be affected. An adequate estimate results in more efficient operation of the Form Driver. You can use the FMS/DIRECTORY/FULL command to find out how much space to allocate.

In the following example from the Sample Application program, workspace is allocated and the FDV$AWKSP routine is called. When the FDV$AWKSP routine is called, the first argument (WORKSPACE) specifies the area of memory to be used for your workspace. In the declaration section of your program, 12 bytes (3 longwords) are allocated to workspace storage. The second argument in the FDV$AWKSP call specifies an estimate of the workspace size (2000 bytes) that you will need to display the largest form in your application.

```
VAR  Workspace : [VOLATILE] ARRAY [1..3] OF INTEGER;

FDV$AWSKP (WORKSPACE,2000);
```

7.9 Precautions for Using FMS

7.9.1 Memory Areas Used Exclusively by FMS

The locations for terminal control area, workspace, and run-time memory-resident form area are used exclusively by FMS. The terminal control area and workspace are attached with the FDV$ATERM and FDV$AWKSP calls and remain allocated until the FDV$DTERM and FDV$DWKSP calls are issued or until the program ends. The run-time memory-resident form area, used in the FDV$READ call, remains allocated until the FDV$DEL call is issued or until the program ends. You never touch the terminal control areas, workspaces, or run-time memory-resident form areas in your program once you have declared them except to pass their addresses to the Form Driver.

7.9.2 Why You Should Use the VOLATILE Attribute

Parameters to the following Form Driver routines should be used with caution:

FDV$ATERM	Attach terminal
FDV$AWKSP	Attach form workspace
FDV$READ	Read form into memory
FDV$SSRV	Specify status reporting variables

For example, once an FDV$SSRV call is issued, the program variables that contain the FMS status and RMS status become volatile and can change at any call point. As a general rule, you should place the status variables in static storage.

In cases where you need both the FMS and RMS statuses, the FDV$STAT routine can be used. Note that only the FDV$STAT and FDV$SSRV calls provide RMS status. With the FDV$STAT routine, you do not have to worry about volatility.

The locations for terminal control area, workspace, run-time memory-resident form area, and status reporting variables must all continue to exist while the Form Driver is using them. They must remain until the terminal control area and workspace are detached, until forms in memory location are deleted, and until the status reporting variables are not used anymore. The variables can be protected by declaring them in static storage with the VOLATILE attribute; otherwise, the compiler might place them in dynamic storage or reuse their storage area.

7.10 Data Conversion

FMS uses only ASCII character strings to display data. All information displayed on the terminal screen, and all information received from the terminal operator, will be represented as ASCII field values. Any manipulation of numeric data requires conversion of ASCII character strings to

numeric data, and conversion of numeric data back to ASCII character strings. In the following discussion of conversion routines, you should assume that the receiving data type can support the largest number that is likely to be generated.

In the Sample Application, the following steps are taken to get a new account balance after writing a check:

```
WITH Registeritem[LastRegisterNumber] DO
     BEGIN
     FDV$RET (FLDVAL := Amtpay, FLDNAM := 'AMTPAY');
     READV (Amtpay, Amount_paid);
     Current_Balance := Current_Balance - Amount_paid;
     TotalPayment := TotalPayment + Amount_paid;

     FDV$PUT (FLDVAL := Integer_to_Text (Current_Balance),
              FLDNAM := 'BALANCE');
```

In this example, the READV procedure is used to convert the character string expression Amtpay to an integer variable Amount_paid, which is used to hold the data item's value. The integer value of the variable Amount_paid is subtracted from the integer value of the variable Current_Balance to produce a new value for Current_Balance. The value of Amount_paid is also added to the integer value of the variable TotalPayment to produce a new value for TotalPayment.

After the data operations have been completed, the function Integer_to_Text converts the integer value of the variable Current_Balance to the corresponding ASCII character expression. (Note that the function Integer_to_Text is not a PASCAL predeclared function; it is a user-defined function created for the Sample Application program.) After the value for balance has been converted to a character expression, it is displayed in a right-justified field 'BALANCE.' The rightmost digit from the program is displayed in the field's rightmost character position. The remaining digits of the character expression are placed to the left of the rightmost digit. If output is longer than the field, FMS truncates on the left. (The Form Driver displays a data length error message (FDV$_DLN) only if you have set FMS Debug mode.)

7.11 Sample Application Program in VAX–11 PASCAL

The FMS Sample Application program (SAMPPAS.PAS) is part of the FMS distribution kit. When FMS is installed, SAMPPAS.PAS is placed in the directory FMS$EXAMPLES. Designed to be a demonstration program and learning tool, the Sample Application shows most of the features provided by FMS. The entire Sample Application program appears at the end of this chapter.

7.11.1 Form Driver Definition Files

The file FDVDEF.PAS is part of the Sample Application program package. When FMS is installed, FDVDEF.PAS is placed in the directory FMS$EXAMPLES. The FDVDEF.PAS file appears after the Sample Application source code.

FDVDEF.PAS contains a variety of codes for the Form Driver routines used in the Sample Application program. Although these codes have been created for use in SAMP.PAS, they can provide you with a helpful starting point as you create definitions for your own application program. The file FDVDEF.PAS includes:

- Predeclared data types
- FMS terminator codes
- Function key terminators returned from the FDV$GET and FDV$WAIT calls
- Form Driver key functions for use with the FDV$DFKBD call
- User action routine (UAR) return codes, which are returned by the UARs to the Form Driver:
  - Field completion UAR return codes
  - Help UAR return codes
  - Function key UAR return codes
- VMS status codes returned when Form Driver routines are called as functions. These codes can be signaled.
- FMS status codes returned when the FDV$STAT routine is called as a function
- Form Driver entry point definitions

7.11.2 Command File for Building the Sample Application Program

The command file for building the Sample Application program includes all the information that you need to compile and link SAMPPAS.PAS. When FMS is installed, the command file is placed in the directory FMS$EXAMPLES.

```
$!     S A M P P A S . C O M
$!
$!     Compile and link the PASCAL version of the FMS V2 Sample Application
$!
$!     The PASCAL source files are:       SAMPPAS.PAS
$!                                        FDVDEF.PAS
$!
$!     SMPVECTOR.OBJ and SMPMEMRES.OBJ were produced by the FMS commands:
$!
$!       $ FMS/VECTOR/OUTPUT=SMPVECTOR   SAMP.FLB
$!       $ FMS/MEMORY/OUTPUT=SMPMEMRES SAMP.FLB/FORM=(HELP_KEYS,HELP_MENU)
$!
$ PASCAL  FDVDEF /ENVIRONMENT
$ PASCAL  SAMPPAS
$ LINK    SAMPPAS, FDVDEF, FMS$EXAMPLES:SMPVECTOR, FMS$EXAMPLES:SMPMEMRES
```

```
[INHERIT ('FDVDEF')] PROGRAM Samp(Sampfile, INPUT, OUTPUT);

{ SAMP -- The FMS V2 Sample Application Program }

{ Data definitions }

{ FMS related }

LABEL   9999;   { End label for error abort }

CONST   Integer_string_length   = 6;
        Numeric_mode            = 0;
        Application_mode        = 1;
        Bell_mode               = 0;
        RegisterSize            = 30;

TYPE    Integer_string  = PACKED ARRAY[1..Integer_string_length] OF CHAR;
        VAR80           = VARYING [80] OF CHAR;
        Fix80           = PACKED ARRAY [1..80] OF CHAR;

{ Account (read in from file) }

        Account_record =  RECORD
          acctno :      PACKED ARRAY [1..5] OF CHAR;
          accdate :     PACKED ARRAY [1..7] OF CHAR;
          last :        PACKED ARRAY [1..20] OF CHAR;
          first :       PACKED ARRAY [1..15] OF CHAR;
          middle :      PACKED ARRAY [1..15] OF CHAR;
          street :      PACKED ARRAY [1..30] OF CHAR;
          city :        PACKED ARRAY [1..20] OF CHAR;
          state :       PACKED ARRAY [1..2] OF CHAR;
          zip :         PACKED ARRAY [1..5] OF CHAR;
          homeph :      PACKED ARRAY [1..10] OF CHAR;
          workph :      PACKED ARRAY [1..10] OF CHAR;
          opw :         PACKED ARRAY [1..12] OF CHAR;
          END;

        Account_as_string = PACKED ARRAY [1..151] OF CHAR;
```

```
{ Deposit data (Read in via FDV$GETAL) }

        Deposit_record =  RECORD
          date :          PACKED ARRAY [1..7] OF CHAR;
          curbal :        PACKED ARRAY [1..6] OF CHAR;
          amt :           PACKED ARRAY [1..6] OF CHAR;
          newbal :        PACKED ARRAY [1..6] OF CHAR;
          memo :          PACKED ARRAY [1..35] OF CHAR;
          END;

        Deposit_as_string = PACKED ARRAY[1..60] OF CHAR;

{ Money:

 Note that all money is kept internally as integers (in cents).
 It is only when the quantities are output that they look like
 dollars, since all the money fields have periods as field
 markers in the right places and they are right justified or
 fixed decimal. }

{ Register data }

        Register_record =  RECORD
          num :           PACKED ARRAY [1..4] OF CHAR;
          date :          PACKED ARRAY [1..7] OF CHAR;
          mempayto :      PACKED ARRAY [1..35] OF CHAR;
          amtdep :        PACKED ARRAY [1..6] OF CHAR;
          amtpay :        PACKED ARRAY [1..6] OF CHAR;
          balance :       PACKED ARRAY [1..6] OF CHAR;
          END;

        Register_as_string = PACKED ARRAY [1..64] OF CHAR;

        Select_Record_Type = (Select_Account, Select_Register);

        Input_Record     = RECORD
            CASE Select_Record_Type OF
                Select_Account: (Account: Account_record);
                Select_Register: (Register: Register_record);
            END;
```

```
{ Other variables }

VAR     Sampfile:       FILE OF Input_Record;
        Sampch:         FILE OF VAR80;

        Workspace :     [VOLATILE]ARRAY [1..3] OF INTEGER; { General workspace }
        CheckwKsp :     [VOLATILE]ARRAY [1..3] OF INTEGER; { Check workspace }
        Tca :           [VOLATILE]ARRAY [1..3] OF INTEGER; { Term Control Area }

        { Storage for memory resident forms }

        Menu_form :     [VOLATILE]ARRAY [1..500] OF INTEGER;
        Check_form :    [VOLATILE]ARRAY [1..750] OF INTEGER;
        Deposit_form :  [VOLATILE]ARRAY [1..500] OF INTEGER;

        Terminator :              INTEGER;  { Terminator returned by FDV }
        Current_Balance :         INTEGER;  { Balance in account, numeric }
        Starting_Balance :        INTEGER;  { Starting balance }
        TotalDeposit :            INTEGER;  { Total deposits made this session }
        TotalPayment :            INTEGER;  { Total checks paid this session }
        Fmsstatus :   [VOLATILE]INTEGER;  { Status for last FDV call }
        Rmsstatus :   [VOLATILE]INTEGER;  { RMS Status for last FDV call }
        LastRegisterNumber :      INTEGER;  { Last number used in register }
        LastCheckNumber :         INTEGER;  { Last check number used }
        CurrentLine :             INTEGER;  { Register line cursor is now on }
        Minwindow :               INTEGER;  { Lowest reg line in scroll area }
        Maxwindow :               INTEGER;  { Highest reg line in scroll area }
        Line :                    Fix80;    { Form image line for check print }
        Password :                Fix80;    { Password from account }
        Uarval:                   Fix80;    { Associated data for UAR }
        Frmnam:                   Fix80;    { Holds form name - RETCX call }
        Hlpnum:                   INTEGER;  { Holds help num. - RETCX call }
        Curpos:                   INTEGER;  { Holds cursor pos. - RETCX call }
        Insovr:                   INTEGER;  { Holds insovr mode - RETCX call }
        Fldtrm:                   INTEGER;  { Field terminator - latest call }
        Size_menu:                INTEGER;  { Gets size of menu form }
        Size_check:               INTEGER;  { Gets size of check form }
        Size_deposit:             INTEGER;  { Gets size of deposit form }

        Account:                  Account_record;

        Deposit:                  Deposit_record;
```

```
        Resisteritem:           ARRAY [1..ResisterSize] OF Resister_record;

        Sample:                 Input_record;



{ FMS terminator codes and FDV entry point definitions are predefined
  in Pascal environment files. }

PROCEDURE Initialize_Account; FORWARD;
PROCEDURE EOF_Cleanup; FORWARD;
PROCEDURE IO_Error_Handler; FORWARD;
PROCEDURE Format_Check; FORWARD;
PROCEDURE MENU; FORWARD;
PROCEDURE EXIT; FORWARD;
PROCEDURE Write_Check; FORWARD;
PROCEDURE Process_Check; FORWARD;
PROCEDURE Finish_Check; FORWARD;
PROCEDURE Print_Check; FORWARD;
PROCEDURE Make_Deposit; FORWARD;
PROCEDURE View_Resister; FORWARD;
PROCEDURE Scroll_Forward; FORWARD;
PROCEDURE Scroll_Backward; FORWARD;
PROCEDURE View_Account; FORWARD;
PROCEDURE Verify_Status; FORWARD;
PROCEDURE GETAL; FORWARD;
PROCEDURE Get_Status; FORWARD;
PROCEDURE Error_report; FORWARD;

[GLOBAL] FUNCTION VALID1: INTEGER; FORWARD;
[GLOBAL] FUNCTION Take15: INTEGER; FORWARD;
[GLOBAL] FUNCTION PASSKY: INTEGER; FORWARD;
[GLOBAL] FUNCTION CHKCHK: INTEGER; FORWARD;
[GLOBAL] FUNCTION RANGE: INTEGER; FORWARD;

FUNCTION Integer_to_Text (Arg: INTEGER): Integer_string;
VAR     Text_value:     VARYING [Integer_string_length] OF CHAR;
BEGIN
WRITEV(Text_value, Arg:Integer_string_length);
Integer_to_Text := Text_value;
END;
```

```
FUNCTION Text_to_Integer (Arg: PACKED ARRAY[I..J:INTEGER] OF CHAR): INTEGER;
VAR     Integer_value:  INTEGER;
BEGIN
READV(Arg, Integer_value);
Text_to_Integer := Integer_value;
END;

FUNCTION Trim(Arg: PACKED ARRAY[I..J: INTEGER] OF CHAR): VAR80;
TYPE    $WORD   = [WORD] 0..65535;
VAR     LEN:    $WORD;
[EXTERNAL] PROCEDURE STR$TRIM(
                        %STDESCR Outstring:     PACKED ARRAY[I..J: INTEGER] OF CHAR;
                        %STDESCR Instring:      [READONLY] PACKED ARRAY[K..L: INTEGER] OF CHAR;
                        VAR      Outlen:        $WORD
                    ); EXTERNAL;
BEGIN
STR$TRIM(Arg, Arg, Len);
Trim := SUBSTR(Arg, 1, Len);
END;



PROCEDURE Initialize_Account;

{ Read from file SAMP.DAT into internal variables.
  Set up the workspace for checks and fill in the check form
  with the account's name, address, and account number. }

LABEL   1000;

BEGIN

{ Open file, get account data}

OPEN (FILE_VARIABLE := Sampfile, FILE_NAME := 'FMS$EXAMPLES:SAMP.DAT',
        HISTORY := READONLY, ERROR := CONTINUE);
IF STATUS(Sampfile) <> 0 THEN IO_Error_Handler;
RESET (Sampfile, ERROR := CONTINUE);
IF STATUS(Sampfile) <> 0 THEN IO_Error_Handler;
READ (Sampfile, Sample, ERROR := CONTINUE);
Account := Sample.Account;
```

```
IF STATUS(Sampfile) <> 0 THEN IO_Error_Handler;

{ Read the remaining records into the check register, counting them.
 The last record has the current balance, and some record has the
 last check number used (not necessarily the last record).}

LastCheckNumber := 0;
LastRegisterNumber := 0;
WHILE LastRegisterNumber < RegisterSize DO
    BEGIN
    IF STATUS(Sampfile) <> 0
    THEN
        BEGIN
        IO_Error_Handler;
        GOTO 1000;
        END
    ELSE
        READ (Sampfile, Sample, ERROR := CONTINUE) ;
    LastRegisterNumber := LastRegisterNumber + 1;
    Registeritem[ LastRegisterNumber ] := Sample.Register;
    IF Registeritem[LastRegisterNumber].Num <> '    '
    THEN
        READV (Registeritem[LastRegisterNumber].Num, LastCheckNumber);
    END;

{ Reached here without hitting end of file.  Ignore remaining records.}

EOF_Cleanup;
1000: END;
```

```
PROCEDURE EOF_Cleanup;

{ Reach here as result of end of file--last record tried didn't read.
  Check for data file in error.
  Take balance from last record read.
  Set session sums to zero to say no activity yet. }

BEGIN
IF LastResisterNumber = 0
THEN
    BEGIN
    WRITELN ('DATA FILE IN ERROR');
    HALT;
    END;

READV(ResisteritemCLastResisterNumber].Balance, Current_Balance);
Startins_balance := Current_Balance;
TotalDeposit := 0;
TotalPayment := 0;

{ Set up the check workspace once so we don't have to do it every time.}

Format_Check;
END;



PROCEDURE IO_Error_Handler;

{ If EOF, close file and cleanup, otherwise use default error handlins. }

BEGIN
IF STATUS(Sampfile) <> 0
THEN
    BEGIN
    CLOSE (Sampfile);
    EOF_Cleanup;
    END;
END;
```

```
PROCEDURE Format_Check;

{ Format account data onto check form in the check workspace. }


BEGIN
  FDV$SWKSP( WKSP := CheckWksp);
  FDV$LOAD( FRMNAM := 'CHECK' );
  With Account DO
      BEGIN
      FDV$PUT( FLDVAL := Trim(First) + ' ' + SUBSTR(Middle,1,1) + '. ' + Trim(Last),
                FLDNAM := 'NAME');
      FDV$PUT( FLDVAL := Street, FLDNAM := 'STREET' );
      FDV$PUT( FLDVAL :=  Trim(City) + ', ' + Trim(State) + ' ' + Trim(Zip),
              FLDNAM :=  'CSZ' );
      FDV$PUT( FLDVAL := Homeph, FLDNAM := 'HOMEPH' );
      FDV$PUT( FLDVAL := Acctno, FLDNAM := 'ACCTNO' );
      FDV$SWKSP( WKSP := Workspace);
      END;
END;



PROCEDURE MENU;

{ Accept inputs from the menu form and dispatch to the
  appropriate routine.  Repeat until option 1 (exit) is
  chosen.  The UARs in the form guarantee that we get back
  only inputs '1'-'5' with the correct terminators.
  Options are:
        1 => Exit
        2 => Write checks
        3 => Make deposit
        4 => View register
        5 => View account data  }
```

```
VAR     Option: PACKED ARRAY [1..1] OF CHAR;
BEGIN
REPEAT
    FDV$CDISP( FRMNAM := 'MENU' );      Verify_Status;
    FDV$GET( Option, Terminator, 'OPTION' );
    CASE Option::CHAR OF
        '1':    EXIT;
        '2':    Write_Check;
        '3':    Make_Deposit;
        '4':    View_Register;
        '5':    View_Account;
        END;
UNTIL Option = '1';
END;



PROCEDURE EXIT;

{ Processing for EXIT menu choice.
  Do nothing but return. }

BEGIN
END;



PROCEDURE Write_Check;

{ Write one or more checks. }

BEGIN

{ Turn on LED 3 on the VT100 during this routine, just to show how.}

FDV$LEDON( LEDNO := 3 );

{ Mark WORKSPACE not displayed so it doesn't show up during a refresh.
  Put up CHECK form from already loaded workspace
  and display current balance. }
```

```
FDV$NDISP;
FDV$SWKSP( WKSP := CheckWksp );
FDV$DISPW;

FDV$PUT( FLDVAL := Integer_to_Text(Current_Balance), FLDNAM := 'BALANCE' );

{ Process checks until a keypad period is read}

Terminator := 0;
WHILE Terminator <> FDV$K_KP_PER DO
    BEGIN
    Process_Check;                              { Process one check}
    Finish_Check;                               { Give options for continuing}
    END;

{ Turn off LED 3 on VT100}

FDV$LEDOF( LEDNO := 3 );
FDV$SWKSP( WKSP := Workspace );

END;



PROCEDURE Process_Check;

{ If input is terminated by kpd period, return with no action
  Else deduct from balance and enter into register.
  Note that a UAR in the form guarantees that the amount of
  the check is always less than or equal to the balance.
  Note that the form function key UAR  allows only kpd period
  as terminator (other than FDV$K_FT_NTR). }

LABEL   1000;
VAR     Amount_paid:    INTEGER;
        Junk:           PACKED ARRAY [1..151] OF CHAR;
BEGIN
FDV$PUT(
FLDVAL := SUBSTR(Integer_to_Text(LastCheckNumber+1),Integer_string_length-1,2),
FLDNAM := 'NUMBER' );
FDV$GETAL( FLDVAL := Junk, FLDTRM := Terminator );
IF Terminator = FDV$K_KP_PER THEN GOTO 1000;
```

```
{ If the check wouldn't fit in the register, don't process, just
  give error message, wait for acknowledgement, and return }

IF LastRegisterNumber = RegisterSize
THEN
    BEGIN
    FDV$PUTL( VAL := 'Register full, cannot enter check' );
    FDV$WAIT;
    GOTO 1000;
    END;


{ Update register array and counters}

LastRegisterNumber := LastRegisterNumber + 1;
LastCheckNumber := LastCheckNumber + 1;

{ Get amount from check.
 Update balance (in memory and on screen) and session sums.
 Transfer form values to register item. }

WITH RegisterItem[LastRegisterNumber] DO
    BEGIN
    FDV$RET( FLDVAL := Amtpay, FLDNAM := 'AMTPAY' );
    READV (Amtpay, Amount_paid);
    Current_Balance := Current_Balance - Amount_paid;
    TotalPayment := TotalPayment + Amount_paid;

    FDV$PUT( FLDVAL := Integer_to_Text(Current_Balance), FLDNAM := 'BALANCE' );
    FDV$RET( FLDVAL := Balance, FLDNAM := 'BALANCE');

    Amtdep := '       ';
    FDV$RET( FLDVAL := Num, FLDNAM := 'NUMBER' );
    FDV$RET( FLDVAL := Date, FLDNAM := 'DATE' );
    FDV$RET( FLDVAL := Mempayto, FLDNAM := 'PAYTO' );   {Note: not from check's MEMO}

    END;

1000: END;
```

```
PROCEDURE Finish_Check;

{ Finish off check processing by giving operator
  three options:
        RETURN  Write another check
        KPD 0   Print the check into file SAMPCH.DAT
        KPD .   Return to menu
  Check to see if check write was aborted by Kpd per.
  If so, then don't give any further choice, just abort.
  Note that form function key UAR allows only the above
  terminators to get through. }

BEGIN
IF Terminator <> FDV$K_KP_PER
THEN
    BEGIN

    { Tell the operator that the check has been paid by overlaying with
     a new form, using the normal workspace, thereby saving the check
     workspace in case another check is to be written. }

    FDV$SWKSP( WKSP := Workspace );
    FDV$DISP( FRMNAM := 'CHECK_DONE' ); Verify_Status;

    { Wait for operator to enter either KPD period, NTR, or KPD zero.
     Print the check as many times as requested.
     (Note that a UAR on the form guarantees that only those terminators
     are accepted).
     Process accordingly. }

    FDV$WAIT( FLDTRM := Terminator );
    WHILE Terminator = FDV$K_KP_0 DO
        BEGIN
        Print_Check;                                { Print the check}
        FDV$WAIT( FLDTRM := Terminator );
        END;

    { If choice is to quit,
      then mark check wksp not displayed so it doesn't appear during refresh
      else mark normal workspace (occupied by CHECK_DONE form) not displayed
        so it doesn't show during refresh and then clear its lines.
```

```
        (Clearing the space occupied by the CHECK_DONE form, lines 20-23,
        is better done by overlaying a blank form to
        avoid having to know the line numbers to clear.) }

    IF Terminator = FDV$K_KP_PER
    THEN
        BEGIN
        FDV$SWKSP( WKSP := CheckWksp );
        FDV$NDISP;
        END
    ELSE
        BEGIN
        FDV$NDISP;
        FDV$CLEAR( LINE := 20, LINECNT := 4 );
        FDV$SWKSP( WKSP := CheckWksp );
        END;

    { Going to write another check now or eventually, so:
      Clear out operator entered fields }

    FDV$PUTD( FLDNAM := 'AMTPAY' );
    FDV$PUTD( FLDNAM := 'MEMO' );
    FDV$PUTD( FLDNAM := 'PAYTO' );
    END;
END;
```

```
PROCEDURE Print_Check;

{ Print the check into the file SAMPCH.DAT
  Use the check workspace, then switch back to the normal wksp
  to keep things clean. }


{ Open check writing file.  Note there's a new version for every check.
 Switch workspaces }

VAR     I, Low_index, High_index, Length:       INTEGER;
        First1, Last1:          PACKED ARRAY [1..2] OF CHAR;

BEGIN
OPEN (FILE_VARIABLE := Sampch, FILE_NAME := 'SAMPCH.DAT',
        HISTORY := NEW);
REWRITE (Sampch);
FDV$SWKSP( WKSP := CheckWksp );

{ Get the top and bottom lines of the check from the named data
 (first two characters). }

FDV$RETDN( NMDNAM := 'FIRST', NMDVAL := First1 ); Verify_Status;
FDV$RETDN( NMDNAM := 'LAST', NMDVAL := Last1 );  Verify_Status;


{ Get lines from form.
 Convert to line printer style.
 Write to file. }

READV (First1, Low_index);
READV (Last1,  High_index);
FOR i := Low_index TO High_index DO
    BEGIN
    FDV$RETFL( LINE := i, VAL := Line, LEN := Length );
    WRITE (Sampch, Line);
    END;
FDV$PUTL( VAL := 'Check written to file' );
CLOSE (Sampch);
FDV$SWKSP( WKSP := Workspace );
END;
```

```
PROCEDURE Make_Deposit;

{ Make a deposit, enter into check register
  Cancel on Keypad period.
  Note that the form function key UAR allows only kpd period. }

{ Put up deposit form with current balance}

LABEL   1000;
VAR     Amount_deposited:       INTEGER;
        Done:                   Fix80;

BEGIN
FDV$CDISP( FRMNAM := 'DEPOSIT' );       Verify_Status;
FDV$PUT( FLDVAL := Integer_to_Text(Current_Balance), FLDNAM := 'CURBAL' );

{ Get deposit amount and memo from operator.
 Abort on Kpd period. }

FDV$GETAL( FLDVAL := Deposit::Deposit_as_string, FLDTRM := Terminator );
IF Terminator = FDV$K_KP_PER THEN GOTO 1000;

{ Have deposit information now. If no room in check register, must abort. }

IF LastRegisterNumber = RegisterSize
THEN
    BEGIN
    FDV$PUTL( VAL := 'Register full, can''t enter deposit' );
    FDV$WAIT;
    GOTO 1000;
    END;

{ Add to balance and session sum.
 Check for overflow (program and form keep only six digits).
 Display new balance.
 Make entry in register. }

READV (Deposit.Amt, Amount_Deposited);
Current_Balance := Current_Balance + Amount_Deposited;
TotalDeposit := TotalDeposit + Amount_Deposited;
IF Current_Balance >= 1000000
THEN
```

```
    BEGIN
    Current_Balance := Current_Balance - 1000000;
    FDV$PUTL(  'Overflow in bank computer, only 6 digits allowed, we keep the rest of the money');
    FDV$WAIT;
    END;
FDV$PUT( FLDVAL := Integer_to_Text( Current_Balance ), FLDNAM := 'NEWBAL' );
LastRegisterNumber := LastRegisterNumber + 1;

WITH Registeritem[LastRegisterNumber] DO
    BEGIN
    Num := '    ';                          { Blank since it's not a check}
    Date := Deposit.Date;
    Mempayto := Deposit.Memo;
    Amtdep := Deposit.Amt;
    Amtpay := '      ';
    FDV$RET( FLDVAL := Balance, FLDNAM := 'NEWBAL' );
    END;

{ Sample of how to keep message texts stored with the form rather
 than in a program.  This is especially useful for multi-lingual
 environments: only the form text and the form named data must
 be changed and nothing in the program.  The trick is to store the
 response text in named data. This is the only example of how to do
 it in this program, but all messages could be stored like this.
 Message intent is: "Deposit made, press RETURN or ENTER to continue." }

FDV$RETDN( NMDNAM := 'DONE', NMDVAL := Done );
FDV$PUTL( VAL := Done );
FDV$WAIT;
1000: END;



PROCEDURE View_Register;

{ View the check register and scroll through it.
  Also display totals for current session. }

VAR     Deposit_Display, Payment_Display:       Integer_string;
        Nscroll:         PACKED ARRAY [1..2] OF CHAR;
        Nscroll_value:   INTEGER;
        Fake:            PACKED ARRAY [1..1] OF CHAR;
```

```
BEGIN

{ Put up register form.
  Check for current session totals overflow. If so, output 'OVRFLO'
  Put out summary of this session into indexed(4) fields. }

FDV$CDISP( FRMNAM := 'REGISTER' );      Verify_Status;
IF TotalDeposit < 1000000 THEN Deposit_Display := Integer_to_Text( TotalDeposit )
                          ELSE Deposit_Display := PAD('OVRFLO', ' ', SIZE(Deposit_Display));
IF TotalPayment < 1000000 THEN Payment_Display := Integer_to_Text( TotalPayment )
                          ELSE Payment_Display := PAD('OVRFLO', ' ', SIZE(Payment_Display));
FDV$PUT( FLDVAL := Integer_to_Text( Starting_balance ), FLDNAM := 'SUMMARY', FLDIDX := 1 );
FDV$PUT( FLDVAL := Deposit_Display, FLDNAM := 'SUMMARY', FLDIDX := 2 );
FDV$PUT( FLDVAL := Payment_Display, FLDNAM := 'SUMMARY', FLDIDX := 3 );
FDV$PUT( FLDVAL := Integer_to_Text( Current_Balance ), FLDNAM := 'SUMMARY', FLDIDX := 4 );

{ Get number of lines in scroll area from form named data (item 1).}

FDV$RETDI( NMDIDX := 1, NMDVAL := Nscroll);     Verify_Status;
READV(Nscroll, Nscroll_Value);

{ Put lines from check register array into scrolled area.
 The window is initially from item 1 up to item
 min(Nscroll, LastRegisterNumber), that is, up to the size of the scrolled
 area or the size of the register, whichever is less. Assume there
 is at least one line (the initial deposit). }

Minwindow := 1;
FDV$PUTSC( FLDNAM := 'NUMBER', FLDVAL := Registeritem[1]::Register_as_string ); { First line}
CurrentLine := 1;                                          { Reg item cursor is on}
WHILE ( CurrentLine < LastRegisterNumber) AND (CurrentLine < Nscroll_Value ) DO
    BEGIN
    CurrentLine := CurrentLine + 1;
    FDV$PFT( FLDTRM := FDV$K_FT_SFW, FLDNAM := 'NUMBER' );
    FDV$PUTSC( FLDNAM := 'NUMBER', FLDVAL := Registeritem[CurrentLine]::Register_as_string );
    END;
Maxwindow := CurrentLine;
```

```
{ Get input from fake field of scrolled line and do what it says:
        kpd . or RETURN/ENTER    => return to menu
        UPARROW or TAB           => scroll forward
        DOWNARROW or BACKSPACE   => scroll backward
        all others               => ignore
 Note that there is no form function key VAR so this routine
 handles all terminators itself (by ignoring illegal ones). }

FDV$GET( FLDVAL := Fake, FLDTRM := Terminator, FLDNAM := 'FAKE' );
WHILE  (Terminator <> FDV$K_FT_NTR) AND (Terminator <> FDV$K_KP_PER) DO
    BEGIN
    IF (Terminator = FDV$K_FT_SFW) OR (Terminator = FDV$K_FT_SNX) THEN Scroll_Forward;
    IF (Terminator = FDV$K_FT_SBK) OR (Terminator = FDV$K_FT_SPR) THEN Scroll_Backward;
    FDV$GET( FLDVAL := Fake, FLDTRM := Terminator, FLDNAM := 'FAKE' );
    END;
END;
```

```
PROCEDURE Scroll_Forward;

{ CurrentLine is the line in the resister that the cursor is on.
  MINWINDOW and MAXWINDOW delimit the part of the resister
  currently displayed in the scrolled area. }


LABEL 1000;

{ If cursor is at the end of the resister, report, and return}

BEGIN
IF CurrentLine = LastResisterNumber
THEN
    BEGIN
    FDV$PUTL( VAL := 'Last line of resister' );
    GOTO 1000;
    END;

{ If cursor not at the last line of a window, just move down
 If cursor is  at the last line of a window,
        move window forward one line,
        write the new last line to the last line of the scrolled area
 Move current line pointer forward. }

IF CurrentLine <> Maxwindow
THEN
    FDV$PFT( FLDTRM := FDV$K_FT_SFW, FLDNAM := 'NUMBER' )
ELSE
    BEGIN
    Minwindow := Minwindow + 1;
    Maxwindow := Maxwindow + 1;
    FDV$PFT( FLDTRM := FDV$K_FT_SFW, FLDNAM := 'NUMBER',
                FLDVAL := Resisteritem[Maxwindow]::Resister_as_string );
    END;
CurrentLine := CurrentLine + 1;
1000: END;
```

```
PROCEDURE Scroll_Backward;

{ CurrentLine is the line in the register that the cursor is on.
  MINWINDOW and MAXWINDOW delimit the part of the register
  currently displayed in the scrolled area. }


LABEL 1000;

BEGIN

{ If the cursor is at the beginning of the register, report, and return}

IF CurrentLine = 1
THEN
    BEGIN
    FDV$PUTL( VAL := 'First line of register' );
    GOTO 1000;
    END;

{ If cursor not at first line of the window, just move up
 If cursor is  at first line of the window,
        move window back one line,
        write the new first line to the first line of the scrolled area
 Move current line pointer back. }

IF CurrentLine <> Minwindow
THEN
    FDV$PFT( FLDTRM := FDV$K_FT_SBK, FLDNAM := 'NUMBER' )
ELSE
    BEGIN
    Minwindow := Minwindow - 1;
    Maxwindow := Maxwindow - 1;
    FDV$PFT( FLDTRM := FDV$K_FT_SBK, FLDNAM := 'NUMBER',
                FLDVAL := Registeritem[Minwindow]::Register_as_string );
    END;
CurrentLine := CurrentLine - 1;
1000: END;
```

```
PROCEDURE View_Account;

{ View the account data.
  If operator knows the secret word, let operator change
  the account data for this session. }

LABEL   1000;
VAR     Password:       PACKED ARRAY [1..12] OF CHAR;
        Junk:           PACKED ARRAY [1..151] OF CHAR;

BEGIN
FDV$CDISP( FRMNAM := 'ACCOUNT_DATA' );
FDV$PUTAL( FRMVAL := Account::Account_as_string );
FDV$PUTD( FLDNAM := 'SECRET');

{ This is not the best way to do protection, just a way of showing
 another FMS feature. At this point, supervisor mode is on, so the
 only input allowed is to the password field.
 If operator doesn't know password, return to menu. }

FDV$GETAL( FLDVAL := Junk, FLDTRM := Terminator );      { Don't care about value now }
IF Terminator = FDV$K_KP_PER THEN GOTO 1000;
FDV$RET( FLDVAL := Password, FLDNAM := 'SECRET' );
IF Account.Opw <> Password THEN GOTO 1000;

{ Allow input from other fields and read from them.
 If read is terminated by Keypad period, don't change account. }

FDV$SPOFF;
GETAL;                  { Read all fields}
FDV$SPON;               { Not really needed, just showing off.}
IF Terminator <> FDV$K_KP_PER
THEN
    BEGIN
    FDV$RETAL( FRMVAL := Account::Account_as_string );
    Format_Check;                       { Update the check workspace}
    END;
1000: END;
```

```
PROCEDURE GETAL;

{ Simulate action of FDV$GETAL, using FDV$GETAF and PFT.  Could
  replace this whole routine with a call on FDV$GETAL, but this shows
  how mainline program can allow same operator freedom of filling in
  fields but still retain control after each or changed field.
  Technique is to read any field, looking only at terminator, then do
  a process field terminator call to do the operator's action.
  This technique can be used with calls on FDV$GET or FDV$GETAF.
  This example starts with a GET on field '*', first field on form. }

LABEL   1000;
VAR     Junk:           PACKED ARRAY [1..151] OF CHAR;
        Fieldname:      PACKED ARRAY [1..6] OF CHAR;
        Fieldindex:     INTEGER;

BEGIN
FDV$GET( FLDVAL := Junk, FLDTRM := Terminator, FLDNAM := '*' );
FDV$RETFN( FLDNAM := Fieldname);        {Get first field's name}
WHILE 1=1 DO
    BEGIN

    { Do any special processing for field FIELDNAME$ at this point.
     ...
     Go to next or previous field or leave form. }

    FDV$PFT( FLDTRM := Terminator );

    { If status is error, then PFT failed because terminator was
     a keypad key, which means return to caller. }

    IF FMSStatus < 0 THEN GOTO 1000;
    IF Terminator = FDV$K_FT_NTR
    THEN
        IF FMSStatus <> 2
        THEN
            GOTO 1000
        ELSE
            BEGIN
            FDV$PUTL( VAL := 'INPUT REQUIRED' );
            FDV$BELL;
            END;
```

```
    { Go set any other field, returning its name}
    FDV$GETAF( FLDVAL := Junk, FLDTRM := Terminator,
               FLDNAM := Fieldname, FLDIDX := Fieldindex);
    END;
1000: END;


PROCEDURE Get_Status;

{ Check FMS status by calling FDV$STAT.
  If not success (>0), print and stop. }

BEGIN
FDV$STAT( STATUS := FMSStatus, IOSTAT := RMSStatus);
IF FMSStatus <= 0 THEN Error_report;    { and never come back}
END;



PROCEDURE Verify_Status;

{ Check FMS status by looking at the status recording variables.}

BEGIN
IF FMSStatus <= 0 THEN Error_report;
END;



PROCEDURE Error_report;

{ There is an error returned in the status variables. Detach the
  terminal to clean up, then print the errors, and stop. }

BEGIN
FDV$DTERM( TCA := Tca );
WRITELN ('FDV ERROR.');
WRITELN ('', 'FMS STATUS:', FMSStatus);
WRITELN ('', 'RMS STATUS:', RMSStatus);
GOTO 9999;
END;
```

```
FUNCTION VALID1;

{ UAR for field validation of any one character field. The
  UAR associated data has in it the legal characters allowed,
  except that blank is not allowed unless it appears before
  the first trailing blank.  For example an assoc. value string
  'aqr' implies that only the letters a, q, and r are allowed.
  A string ' aqr' means that blank is acceptable in addition
  to a, q, and r. Note that this routine is case sensitive
  (that is, it checks for correct case).  You can get around
  case sensitivity by using the force upper case field attribute
  and putting only capitals into the UAR associated value
  string. This routine can be used with any form and field since
  it determines the context for itself. }

VAR     Fldname:        PACKED ARRAY [1..31] OF CHAR;
        FValue:         PACKED ARRAY [1..1] OF CHAR;
        Fieldindex:     INTEGER;

BEGIN

{ Retrieve context: we will ignore TCA address, WKSP address, FRMNAM,
        CURPOS, FLDTRM, and INSOVR, using only UARVAL, and only the
        initial, non-blank characters of it.
 Retrieve field name and index. Retrieve field value. }

FDV$RETCX( TCA := Tca, WKSP := Workspace, FRMNAM := Frmnam, UARVAL := Uarval,
    CURPOS := Curpos, FLDTRM := Fldtrm, INSOVR := Insovr, HLPNUM := Hlpnum );
FDV$RETFN( FLDNAM := Fldname, FLDIDX := Fieldindex);
FDV$RET( FLDVAL := Fvalue, FLDNAM := Fldname, FLDIDX := Fieldindex);

{ To be valid, FVALUE must occur in the string UARVAL. }

IF INDEX( Uarval, Fvalue) > 0
THEN
    VALID1 := FDV$K_UVAL_SUC                    {Success}
ELSE
    BEGIN
    FDV$PUTL( VAL := 'Illegal value' );
    VALID1 := FDV$K_UVAL_FAIL;
    END;
END;
```

```
FUNCTION Take15;

{ Function Key User Action Routine for the MENU form of SAMP.
  Convert Keypad 1-5 into field values 1-5.
  Convert Keypad period into field value 1.
  Reject all other function keys with error message. }

VAR     Mapping:        PACKED ARRAY [1..1] OF CHAR;

BEGIN

{ Retrieve context: we will ignore TCA address, WKSP address, FRMNAM,
        UARVAL, CURPOS and INSOVR, using only FLDTRM. }

FDV$RETCX( TCA := Tca, WKSP := Workspace, FRMNAM := Frmnam, UARVAL := Uarval,
    CURPOS := Curpos, FLDTRM := Fldtrm, INSOVR := Insovr, HLPNUM := Hlpnum );

{ Do the conversion, displaying the value converted if found.
 Reject if not one of the expected terminators. }

Mapping := ' ';
IF Fldtrm = FDV$K_KP_1 THEN Mapping := '1';
IF Fldtrm = FDV$K_KP_2 THEN Mapping := '2';
IF Fldtrm = FDV$K_KP_3 THEN Mapping := '3';
IF Fldtrm = FDV$K_KP_4 THEN Mapping := '4';
IF Fldtrm = FDV$K_KP_5 THEN Mapping := '5';
IF Fldtrm = FDV$K_KP_PER THEN Mapping := '1';
IF Mapping <> ' '
THEN
    BEGIN
    FDV$PUT( FLDVAL := Mapping, FLDNAM := 'OPTION' );
            { Treat as if it is RETURN}
    Take15 := FDV$K_UKEY_NTR;
    END
ELSE
    BEGIN
    FDV$PUTL( VAL := 'Illegal function key' );
    FDV$SIGOP;
   { Just ignore it now}
    Take15 := FDV$K_UKEY_SUC;
    END;
END;
```

```
FUNCTION PASSKY;

{ General function key var to pass only those from the (small) list
  in the var associated value string and reject all others.
  The list is of the form: n <oneblank> n <oneblank> ... n <manyblanks>
  For example the string '110 112' would accept keypad period and
  keypad zero but no other function keys. }

LABEL   1000;
VAR     Nexttrm:        INTEGER;
        NonBlank:       INTEGER;
        NextBlank:      INTEGER;

BEGIN

{ Retrieve context: we will ignore TCA address, WKSP address, FRMNAM,
        INSOVR, and CURPOS, using only FLDTRM and UARVAL. }

FDV$RETCX( TCA := Tca, WKSP := Workspace, FRMNAM := Frmnam, UARVAL := Uarval,
    CURPOS := Curpos, FLDTRM := Fldtrm, INSOVR := Insovr, HLPNUM := Hlpnum );

{ Break up the list into numbers.  Check each against the actual
 terminator. If terminator found in list, return success. }

Nonblank := 1;          { Beginning of string}
WHILE (Uarval[Nonblank] <> ' ') AND (Nonblank <= 80) DO
    BEGIN
    Nextblank := INDEX( SUBSTR(Uarval, Nonblank, LENGTH(Uarval) - Nonblank + 1),
                        ' ');
    IF Nextblank = 0    THEN Nextblank := 80
                        ELSE Nextblank := Nextblank + Nonblank - 1;
    READV (SUBSTR(Uarval, Nonblank, Nextblank-Nonblank), Nexttrm);
    IF Fldtrm = Nexttrm
    THEN
        BEGIN
        PASSKY := FDV$K_UKEY_TRM;       {Pass key to application}
        GOTO 1000;
        END;
    Nonblank := Nextblank + 1;
    END;
PASSKY := FDV$K_UKEY_ERR;               {Let FDV do the beeping}
1000: END;
```

```
FUNCTION CHKCHK;


{ UAR for SAMP CHECK form.  Makes sure that the check amount is
  less than or equal to the current balance.  If not, complain and
  change video attributes on balance field so the potential bouncer
  can see what there is to work with. }

VAR     Balance, Amtpay:        Integer_string;
        BlinkBold:              INTEGER;

BEGIN
FDV$RET( FLDVAL := Balance, FLDNAM := 'BALANCE' );
FDV$RET( FLDVAL := Amtpay, FLDNAM := 'AMTPAY' );
IF Text_to_Integer( Balance )  >= Text_to_Integer( Amtpay )
THEN
    BEGIN
    CHKCHK := FDV$K_UVAL_SUC;
    BlinkBold := -1;                        {Restore to original}
    FDV$AFVA( VIDEO := BlinkBold, FLDNAM := 'BALANCE' );
    END
ELSE
    BEGIN
    CHKCHK := FDV$K_UVAL_FAIL;
    BlinkBold := 3;                         {Make it very visible}
    FDV$AFVA( VIDEO := BlinkBold, FLDNAM := 'BALANCE' );
    FDV$PUTL( VAL := 'Your balance doesn''t cover that much, reenter amount');
    END;
END;
```

```
FUNCTION RANGE;


{ General purpose UAR to check the range of any numeric item.  The
  associated UAR data must have one of the four forms:
        L,U<space><message>
        ,U<space><message>
        L,<space><message>
        ,<space><message>
  where L is lower bound, U is upper bound, and <message> is a
  optional error message in case the field value is out of bounds.
  If one of the bounds isn't given, it isn't checked for.  If neither
  bound is given, nothing is checked, everything succeeds.  If the
  UAR value doesn't have a comma, a FDV$_UAR error message is returned
  to the calling program by the FDV so the form designer has to go
  back and do it right. If no <message> is given, a simple
  "out of range U:L" message is given to the hapless operator.
  This UAR can work with any form and numeric field since it gets
  context itself.  Care must be taken with fields using field marker
  periods since those periods are not returned to the program. }

LABEL   1000, 2000;
VAR     N, Num:             INTEGER;
        Name:               PACKED ARRAY [1..31] OF CHAR;
        Number:             PACKED ARRAY [1..132] OF CHAR;
        Fieldindex:         INTEGER;
        Comma:              INTEGER;
        Blank:              INTEGER;

BEGIN

{ Get context which yields associated data value (ignore other stuff).
 Get current field name and index.
 Get field value. }

FDV$RETCX( TCA := Tca, WKSP := Workspace, FRMNAM := Frmnam, UARVAL := Uarval,
    CURPOS := Curpos, FLDTRM := Fldtrm, INSOVR := Insovr, HLPNUM := Hlpnum );
FDV$RETFN( FLDNAM := Name, FLDIDX := Fieldindex);
FDV$RET( FLDVAL := Number, FLDNAM := Name, FLDIDX := Fieldindex );
READV (Number, Num);

{ Find comma and blank delimiters. Check for lower bound. }
```

```
Comma := INDEX( Uarval, ',' );
Blank := INDEX(SUBSTR(Uarval, Comma+1, LENGTH(Uarval)-Comma), ' ');
IF Blank <> 0 THEN Blank := Comma + Blank ELSE Blank := Comma + 1;
IF Comma = 0
THEN
    BEGIN
    RANGE := 0;          { Illegal UARVAL string, FDV returns error}
    GOTO 2000;
    END;
IF Comma <> 1
THEN
    BEGIN
    READV ( SUBSTR(Uarval, 1, Comma-1), N);
    IF Num < N THEN GOTO 1000;
    END;

{ Check for upper bound}

IF Blank <> Comma + 1
THEN
    BEGIN
    READV (SUBSTR(Uarval, Comma+1, Blank-Comma-1), N);
    IF Num > N THEN GOTO 1000;
    END;

{ Passed both tests successfully, return success for UAR value}

RANGE := FDV$K_UVAL_SUC;
GOTO 2000;

{ Error in one of the bounds.
 Give error message: either from the UARVAL or make one up. }

1000: IF Uarval[Blank + 1] <> ' '
      THEN
           FDV$PUTL( VAL := SUBSTR( Uarval, Blank + 1, 80-Blank ) )
      ELSE
           FDV$PUTL( VAL := 'Field value out of bounds. Must be in range' {+ ' "' +
                        SUBSTR( Uarval, 1, Blank - 1 ) + '".'} );
FDV$SIGOP;                {Beep, too.}
RANGE := FDV$K_UVAL_FAIL;
2000: END;
```

```
BEGIN

{ Main routine of SAMP }

{ Initialize FMS
   Attach default terminal
   Attach normal and check workspaces (order important for help
       and refresh during CHECK/CHECK_DONE time--try switching and see).
   Open form library, attach to channel 1
   Set keypad mode to application
   Set signal mode to bell (default, but it's fun to do). }

FDV$ATERM( TCA := Tca, SIZE := 12, CHANNEL := 2 );      Get_Status;
FDV$AWKSP( WKSP := CheckWksp, SIZE := 2000 );           Get_Status;
FDV$AWKSP( WKSP := Workspace, SIZE := 2000 );           Get_Status;
FDV$LOPEN( 'FMS$EXAMPLES:SAMP', 1 );                    Get_Status;
FDV$SPADA( MODE := Application_mode );
FDV$SSIGQ( SIGMD := Bell_mode );

{ Set all future calls to return status to the two status recording
 variables FMSStatus and RMSStatus without having to call the
 the FDV$STAT routine. }

FDV$SSRV( STATUS := FMSStatus, IOSTAT := RMSStatus );

{ Read in a few forms from the form library onto the dynamic
 resident form list.  You may be able to detect the difference
 in the form to form access times for those forms which have to be
 accessed from the form library on disk and those forms which are
 on the dynamic or static memory resident form list.  See the
 installation notes for this program (the LINK command) to see
 which forms are on the static memory resident form list. }

FDV$READ( FRMNAM := 'MENU', MEMLOC := Menu_Form, SIZE := 2000, FRMSIZ := Size_menu );
FDV$READ( FRMNAM := 'CHECK', MEMLOC := Check_Form, SIZE := 3000, FRMSIZ := Size_check );
FDV$READ( FRMNAM := 'DEPOSIT', MEMLOC := Dposit_Form, SIZE := 2000, FRMSIZ := Size_dposit );
```

```
{ Initialize account information }

Initialize_Account;

{ Put up welcome form, wait for response}

FDV$CDISP( FRMNAM := 'WELCOME' );                          Verify_Status;
FDV$WAIT;

{ Process all menu requests}

MENU;

{ Clean up and leave:
   Close form library.
   Reset keypad to numeric.
   Delete a form from dynamic mem. res. form list just to show how.
   Detach workspaces (not really necessary since DTERM would do it).
   Detach terminal. }

FDV$LCLOS;
FDV$SPADA( MODE := Numeric_mode );
FDV$DEL( FRMNAM := 'MENU' );
FDV$DWKSP( WKSP := Workspace );
FDV$DWKSP( WKSP := CheckWksp );
FDV$DTERM( TCA := Tca );
9999:;
END.
```

```
(*****************************************************************************)
(* Pascal environment for FMS Form Driver                                    *)
(*****************************************************************************)


MODULE FDVDEF ;

[HIDDEN] TYPE   (**** Pre-declared data types ****)
        $BYTE = [BYTE] -128..127;
        $WORD = [WORD] -32768..32767;
        $QUAD = [QUAD,UNSAFE] RECORD
                L0:UNSIGNED; L1:INTEGER; END;
        $OCTA = [OCTA,UNSAFE] RECORD
                L0,L1,L2:UNSIGNED; L3:INTEGER; END;
        $UBYTE = [BYTE] 0..255;
        $UWORD = [WORD] 0..65535;
        $UQUAD = [QUAD,UNSAFE] RECORD
                L0,L1:UNSIGNED; END;
        $UOCTA = [OCTA,UNSAFE] RECORD
                L0,L1,L2,L3:UNSIGNED; END;
        $PACKED_DEC = [BIT(4),UNSAFE] 0..15;
        $DEFTYP = [UNSAFE] INTEGER;
        $DEFPTR = [UNSAFE] $DEFTYP;
        $BIT1 = [BIT(1),UNSAFE] BOOLEAN;
        $BIT2 = [BIT(2),UNSAFE] 0..3;
        $BIT3 = [BIT(3),UNSAFE] 0..7;
        $BIT4 = [BIT(4),UNSAFE] 0..15;
        $BIT5 = [BIT(5),UNSAFE] 0..31;
        $BIT6 = [BIT(6),UNSAFE] 0..63;
        $BIT7 = [BIT(7),UNSAFE] 0..127;
        $BIT8 = [BIT(8),UNSAFE] 0..255;
        $BIT9 = [BIT(9),UNSAFE] 0..511;
        $BIT10 = [BIT(10),UNSAFE] 0..1023;
        $BIT11 = [BIT(11),UNSAFE] 0..2047;
        $BIT12 = [BIT(12),UNSAFE] 0..4095;
        $BIT13 = [BIT(13),UNSAFE] 0..8191;
        $BIT14 = [BIT(14),UNSAFE] 0..16383;
        $BIT15 = [BIT(15),UNSAFE] 0..32767;
        $BIT16 = [BIT(16),UNSAFE] 0..65535;
        $BIT17 = [BIT(17),UNSAFE] 0..131071;
        $BIT18 = [BIT(18),UNSAFE] 0..262143;
        $BIT19 = [BIT(19),UNSAFE] 0..524287;
```

```
        $BIT20 = [BIT(20),UNSAFE] 0..1048575;
        $BIT21 = [BIT(21),UNSAFE] 0..2097151;
        $BIT22 = [BIT(22),UNSAFE] 0..4194303;
        $BIT23 = [BIT(23),UNSAFE] 0..8388607;
        $BIT24 = [BIT(24),UNSAFE] 0..16777215;
        $BIT25 = [BIT(25),UNSAFE] 0..33554431;
        $BIT26 = [BIT(26),UNSAFE] 0..67108863;
        $BIT27 = [BIT(27),UNSAFE] 0..134217727;
        $BIT28 = [BIT(28),UNSAFE] 0..268435455;
        $BIT29 = [BIT(29),UNSAFE] 0..536870911;
        $BIT30 = [BIT(30),UNSAFE] 0..1073741823;
        $BIT31 = [BIT(31),UNSAFE] 0..2147483647;
        $BIT32 = [BIT(32),UNSAFE] UNSIGNED;

(*** MODULE FDVDEF ***)

(*************************)
(* FMS terminator codes *)
(*************************)
CONST   FDV$K_FT_NTR     =  0;  (* Enter (i.e. end GETs) *)
        FDV$K_FT_NXT     =  1;  (* Next field *)
        FDV$K_FT_PRV     =  2;  (* Previous field *)
        FDV$K_FT_ATB     =  3;  (* Automatically move to next field *)
        FDV$K_FT_XBK     =  4;  (* Exit scrolled area backward *)
        FDV$K_FT_XFW     =  5;  (* Exit scrolled area forward *)
        FDV$K_FT_SNX     =  6;  (* Scroll forward to next field  *)
        FDV$K_FT_SPR     =  7;  (* Scroll backward to previous field *)
        FDV$K_FT_SFW     =  8;  (* Scroll forward *)
        FDV$K_FT_SBK     =  9;  (* Scroll backward *)
        FDV$K_FT_ILG_NXT = 11;  (* Illegal context for next field  *)
        FDV$K_FT_ILG_PRV = 12;  (* Illegal context for previous field  *)
        FDV$K_FT_ILG_ATB = 13;  (* Illegal context for auto move to next fld *)
        FDV$K_FT_ILG_XBK = 14;  (* Illegal context for exit sc area backward *)
        FDV$K_FT_ILG_XFW = 15;  (* Illegal context for exit sc area forward *)
        FDV$K_FT_ILG_SFW = 16;  (* Illegal context for scroll forward *)
        FDV$K_FT_ILG_SBK = 17;  (* Illegal context for scroll backward *)
(*********************************************************)
(* Function Key terminators returned from GETs and WAIT *)
(* Also used as FDV keycodes for use with DFKBD.        *)
(*********************************************************)
        FDV$K_AR_UP      =   99;
        FDV$K_AR_DOWN    =  100;
```

```
FDV$K_AR_LEFT = 101;
FDV$K_AR_RIGHT = 102;
FDV$K_PF_1 = 103;
FDV$K_PF_2 = 104;
FDV$K_PF_3 = 105;
FDV$K_PF_4 = 106;
FDV$K_KP_NTR = 107;
FDV$K_KP_COM = 108;
FDV$K_KP_HYP = 109;
FDV$K_KP_PER = 110;
FDV$K_KP_0 = 112;
FDV$K_KP_1 = 113;
FDV$K_KP_2 = 114;
FDV$K_KP_3 = 115;
FDV$K_KP_4 = 116;
FDV$K_KP_5 = 117;
FDV$K_KP_6 = 118;
FDV$K_KP_7 = 119;
FDV$K_KP_8 = 120;
FDV$K_KP_9 = 121;
FDV$K_GAR_UP = 227;
FDV$K_GAR_DOWN = 228;
FDV$K_GAR_RIGHT = 229;
FDV$K_GAR_LEFT = 230;
FDV$K_GPF_1 = 231;
FDV$K_GPF_2 = 232;
FDV$K_GPF_3 = 233;
FDV$K_GPF_4 = 234;
FDV$K_GKP_NTR = 235;
FDV$K_GKP_COM = 236;
FDV$K_GKP_HYP = 237;
FDV$K_GKP_PER = 238;
FDV$K_GKP_0 = 240;
FDV$K_GKP_1 = 241;
FDV$K_GKP_2 = 242;
FDV$K_GKP_3 = 243;
FDV$K_GKP_4 = 244;
FDV$K_GKP_5 = 245;
FDV$K_GKP_6 = 246;
FDV$K_GKP_7 = 247;
FDV$K_GKP_8 = 248;
FDV$K_GKP_9 = 249;
```

```
(*********************************************)
(* FDV KeyFunctions. For use in DFKBD call. *)
(*********************************************)
        FDV$K_KF_GOLD   =   1;
        FDV$K_KF_RESET  =   2;
        FDV$K_KF_CRSLF  =   3;
        FDV$K_KF_CRSRT  =   4;
        FDV$K_KF_DLCHR  =   5;
        FDV$K_KF_DLFLD  =   6;
        FDV$K_KF_INS    =   7;
        FDV$K_KF_OVR    =   8;
        FDV$K_KF_RFRSH  =   9;
        FDV$K_KF_HELP   =  10;
        FDV$K_KF_NXT    =  11;
        FDV$K_KF_PRV    =  12;
        FDV$K_KF_NTR    =  13;
        FDV$K_KF_SBK    =  14;
        FDV$K_KF_SFW    =  15;
        FDV$K_KF_XBK    =  16;
        FDV$K_KF_XFW    =  17;
        FDV$K_KF_DFLT   =  -1;
        FDV$K_KF_NONE   =   0;
(*****************************************************************)
(* UAR return codes. These codes are returned by UAR to FDV. *)
(*****************************************************************)
(* Field completion return codes *)
(*********************************)
        FDV$K_UVAL_SUC =  1000; (* Field completion success *)
        FDV$K_UVAL_FAIL=  1001; (* Field completion failure *)
        FDV$K_UVAL_END =  1002; (* Field completion suc-stop UARs *)
(*************************)
(* Help UAR return codes *)
(*************************)
        FDV$K_UHELP_NO =  2000; (* No help given, try next step *)
        FDV$K_UHELPED  =  2001; (* Help given, continue sequence *)
        FDV$K_UHELP_ALL=  2002; (* Help given, repeat UAR *)
(***********************************)
(* Function Key UAR return codes *)
(***********************************)
        FDV$K_UKEY_ERR =  3000; (* Fn Key failure, FDV signals *)
        FDV$K_UKEY_TRM =  3001; (* Fn Key success, normal f.K. *)
        FDV$K_UKEY_NXT =  3002; (* Fn Key succ, treat as NEXT *)
```

```
        FDV$K_UKEY_NTR =  3003; (* Fn Key succ, treat as ENTER *)
        FDV$K_UKEY_SUC =  3004; (* Fn Key succ, ignore *)
(*************************************************************************)
(* FDV status codes returned when FDV$... routines are called as functions. *)
(* These codes are VMS status codes and can be signalled.  They correspond  *)
(* one-to-one with the FMS status codes retrievable from FDV$STAT.          *)
(*************************************************************************)
        FDV$_SUC = 2719889 ;
        FDV$_INC = 2719897 ;
        FDV$_MOD = 2719905 ;
        FDV$_IMP = 2719922 ;
        FDV$_FSP = 2719930 ;
        FDV$_IOL = 2719938 ;
        FDV$_FLB = 2719946 ;
        FDV$_ICH = 2719954 ;
        FDV$_FCH = 2719962 ;
        FDV$_FRM = 2719970 ;
        FDV$_FNM = 2719978 ;
        FDV$_LIN = 2719986 ;
        FDV$_FLD = 2719994 ;
        FDV$_NOF = 2720002 ;
        FDV$_DSP = 2720010 ;
        FDV$_NSC = 2720018 ;
        FDV$_DNM = 2720026 ;
        FDV$_DLN = 2720034 ;
        FDV$_UTR = 2720042 ;
        FDV$_IOR = 2720050 ;
        FDV$_IFN = 2720058 ;
        FDV$_ARG = 2720066 ;
        FDV$_INI = 2720074 ;
        FDV$_STR = 2720082 ;
        FDV$_IVM = 2720090 ;
        FDV$_FVM = 2720098 ;
        FDV$_ITT = 2720106 ;
        FDV$_TCA = 2720114 ;
        FDV$_STA = 2720122 ;
        FDV$_WID = 2720130 ;
        FDV$_NFL = 2720138 ;
        FDV$_IBF = 2720146 ;
        FDV$_NDS = 2720154 ;
        FDV$_UDP = 2720162 ;
        FDV$_UAR = 2720170 ;
```

```
        FDV$_UNF = 2720178 ;
        FDV$_CAN = 2720194 ;
        FDV$_KIF = 2720202 ;
        FDV$_KEX = 2720210 ;
        FDV$_KTW = 2720218 ;
        FDV$_KIL = 2720226 ;
        FDV$_TMO = 2720234 ;
        FDV$_LLI = 2720242 ;
        FDV$_VAL = 2720250 ;
        FDV$_IFU = 2720258 ;
        FDV$_SYS = 2720266 ;

(*****************************************************************************)
(* FMS status codes returned when FDV$STAT routine is called.                *)
(*****************************************************************************)
(* Success codes. *)

        FDV$K_SUC = 1;
        FDV$K_INC = 2;
        FDV$K_MOD = 3;

(* Failure code *)

        FDV$K_IMP = -2;
        FDV$K_FSP = -3;
        FDV$K_IOL = -4;
        FDV$K_FLB = -5;
        FDV$K_ICH = -6;
        FDV$K_FCH = -7;
        FDV$K_FRM = -8;
        FDV$K_FNM = -9;
        FDV$K_LIN = -10;
        FDV$K_FLD = -11;
        FDV$K_NOF = -12;
        FDV$K_DSP = -13;
        FDV$K_NSC = -14;
        FDV$K_DNM = -15;
        FDV$K_DLN = -16;
        FDV$K_UTR = -17;
        FDV$K_IOR = -18;
        FDV$K_IFN = -19;
        FDV$K_ARG = -20;
```

```
        FDV$K_INI = -21;
        FDV$K_STR = -22;
        FDV$K_FVM = -23;
        FDV$K_IVM = -24;
        FDV$K_ITT = -25;
        FDV$K_TCA = -26;
        FDV$K_STA = -27;
        FDV$K_WID = -28;
        FDV$K_NFL = -29;
        FDV$K_IBF = -30;
        FDV$K_NDS = -31;
        FDV$K_UDP = -33;
        FDV$K_UAR = -34;
        FDV$K_UNF = -35;
        FDV$K_CAN = -39;
        FDV$K_KIF = -40;
        FDV$K_KEX = -41;
        FDV$K_KTW = -42;
        FDV$K_KIL = -43;
        FDV$K_TMO = -44;
        FDV$K_LLI = -45;
        FDV$K_VAL = -47;
        FDV$K_IFU = -48;
        FDV$K_SYS = -49;


(* Form Driver Entry point definitions *)

[ASYNCHRONOUS] FUNCTION FDV$AFCX (
        insovr : INTEGER;
        curpos : INTEGER;
        fldnam : [CLASS_S] PACKED ARRAY [$13..$u3:INTEGER] OF CHAR := %IMMED 0;
        fldidx : INTEGER := %IMMED 0) : INTEGER; EXTERNAL;

[ASYNCHRONOUS] FUNCTION FDV$AFVA (
        VAR video : [VOLATILE]INTEGER;
        fldnam : [CLASS_S] PACKED ARRAY [$12..$u2:INTEGER] OF CHAR := %IMMED 0;
        fldidx : INTEGER := %IMMED 0) : INTEGER; EXTERNAL;
```

```
[ASYNCHRONOUS] FUNCTION FDV$ATERM (
        VAR tca : [VOLATILE]ARRAY [$11..$u1:INTEGER] OF INTEGER;
        size : INTEGER := %IMMED 0;
        channel : INTEGER := %IMMED 0;
        terminal : INTEGER := %IMMED 0) : INTEGER; EXTERNAL;

[ASYNCHRONOUS] FUNCTION FDV$AWKSP (
        VAR wksp : [VOLATILE]ARRAY [$11..$u1:INTEGER] OF INTEGER;
        size : INTEGER) : INTEGER; EXTERNAL;

[ASYNCHRONOUS] FUNCTION FDV$BELL : INTEGER; EXTERNAL;

[ASYNCHRONOUS] FUNCTION FDV$CDISP (
        frmnam : [CLASS_S] PACKED ARRAY [$11..$u1:INTEGER] OF CHAR;
        offset : INTEGER := %IMMED 0) : INTEGER; EXTERNAL;

[ASYNCHRONOUS] FUNCTION FDV$CLEAR (
        line : INTEGER;
        linecnt : INTEGER := %IMMED 0) : INTEGER; EXTERNAL;

[ASYNCHRONOUS] FUNCTION FDV$DEL (
        frmnam : [CLASS_S] PACKED ARRAY [$11..$u1:INTEGER] OF CHAR) : INTEGER; EXTERNAL;

[ASYNCHRONOUS] FUNCTION FDV$DISP (
        frmnam : [CLASS_S] PACKED ARRAY [$11..$u1:INTEGER] OF CHAR;
        offset : INTEGER := %IMMED 0) : INTEGER; EXTERNAL;

[ASYNCHRONOUS] FUNCTION FDV$DISPW (
        offset : INTEGER := %IMMED 0) : INTEGER; EXTERNAL;

[ASYNCHRONOUS] FUNCTION FDV$DTERM (
        VAR tca : [VOLATILE]ARRAY [$11..$u1:INTEGER] OF INTEGER := %IMMED 0) : INTEGER; EXTERNAL;

[ASYNCHRONOUS] FUNCTION FDV$DWKSP (
        VAR wksp : [VOLATILE]ARRAY [$11..$u1:INTEGER] OF INTEGER) : INTEGER; EXTERNAL;

[ASYNCHRONOUS] FUNCTION FDV$GET (
        VAR fldval : [CLASS_S] PACKED ARRAY [$11..$u1:INTEGER] OF CHAR;
        VAR fldtrm : [VOLATILE]INTEGER;
        fldnam : [CLASS_S] PACKED ARRAY [$13..$u3:INTEGER] OF CHAR;
        fldidx : INTEGER := %IMMED 0) : INTEGER; EXTERNAL;
```

```
[ASYNCHRONOUS] FUNCTION FDV$GETAF (
        VAR fldval : [CLASS_S] PACKED ARRAY [$11..$u1:INTEGER] OF CHAR;
        VAR fldtrm : [VOLATILE]INTEGER;
        VAR fldnam : [CLASS_S] PACKED ARRAY [$13..$u3:INTEGER] OF CHAR;
        VAR fldidx : [VOLATILE]INTEGER := %IMMED 0) : INTEGER; EXTERNAL;

[ASYNCHRONOUS] FUNCTION FDV$GETAL (
        VAR fldval : [CLASS_S] PACKED ARRAY [$11..$u1:INTEGER] OF CHAR;
        VAR fldtrm : [VOLATILE]INTEGER := %IMMED 0;
        fldnam : [CLASS_S] PACKED ARRAY [$13..$u3:INTEGER] OF CHAR := %IMMED 0;
        fldidx : INTEGER := %IMMED 0) : INTEGER; EXTERNAL;

[ASYNCHRONOUS] FUNCTION FDV$GETDL (
        VAR val : [CLASS_S] PACKED ARRAY [$11..$u1:INTEGER] OF CHAR;
        VAR fldtrm : [VOLATILE]INTEGER := %IMMED 0;
        line : INTEGER := %IMMED 0) : INTEGER; EXTERNAL;

[ASYNCHRONOUS] FUNCTION FDV$GETSC (
        fldnam : [CLASS_S] PACKED ARRAY [$11..$u1:INTEGER] OF CHAR;
        VAR fldval : [CLASS_S] PACKED ARRAY [$12..$u2:INTEGER] OF CHAR;
        VAR fldtrm : [VOLATILE]INTEGER := %IMMED 0) : INTEGER; EXTERNAL;

[ASYNCHRONOUS] FUNCTION FDV$LCHAN (
        channel : INTEGER) : INTEGER; EXTERNAL;

[ASYNCHRONOUS] FUNCTION FDV$LCLOS : INTEGER; EXTERNAL;

[ASYNCHRONOUS] FUNCTION FDV$LEDOF (
        ledno : INTEGER) : INTEGER; EXTERNAL;

[ASYNCHRONOUS] FUNCTION FDV$LEDON (
        ledno : INTEGER) : INTEGER; EXTERNAL;

[ASYNCHRONOUS] FUNCTION FDV$LEN (
        VAR flen : [VOLATILE]INTEGER;
        VAR fldnam : [CLASS_S] PACKED ARRAY [$12..$u2:INTEGER] OF CHAR;
        fldidx : INTEGER := %IMMED 0) : INTEGER; EXTERNAL;

[ASYNCHRONOUS] FUNCTION FDV$LOAD (
        frmnam : [CLASS_S] PACKED ARRAY [$11..$u1:INTEGER] OF CHAR) : INTEGER; EXTERNAL;
```

```
[ASYNCHRONOUS] FUNCTION FDV$LOPEN (
        filspc : [CLASS_S] PACKED ARRAY [$l1..$u1:INTEGER] OF CHAR;
        channel : INTEGER := %IMMED 0) : INTEGER; EXTERNAL;

[ASYNCHRONOUS] FUNCTION FDV$NDISP : INTEGER; EXTERNAL;

[ASYNCHRONOUS] FUNCTION FDV$PFT (
        fldtrm : INTEGER;
        fldnam : [CLASS_S] PACKED ARRAY [$l2..$u2:INTEGER] OF CHAR := %IMMED 0;
        fldval : [CLASS_S] PACKED ARRAY [$l3..$u3:INTEGER] OF CHAR := %IMMED 0;
        fldidx : INTEGER := %IMMED 0) : INTEGER; EXTERNAL;

[ASYNCHRONOUS] FUNCTION FDV$PUT (
        fldval : [CLASS_S] PACKED ARRAY [$l1..$u1:INTEGER] OF CHAR;
        fldnam : [CLASS_S] PACKED ARRAY [$l2..$u2:INTEGER] OF CHAR;
        fldidx : INTEGER := %IMMED 0) : INTEGER; EXTERNAL;

[ASYNCHRONOUS] FUNCTION FDV$PUTAL (
        frmval : [CLASS_S] PACKED ARRAY [$l1..$u1:INTEGER] OF CHAR) : INTEGER; EXTERNAL;

[ASYNCHRONOUS] FUNCTION FDV$PUTD (
        fldnam : [CLASS_S] PACKED ARRAY [$l1..$u1:INTEGER] OF CHAR;
        fldidx : INTEGER := %IMMED 0) : INTEGER; EXTERNAL;

[ASYNCHRONOUS] FUNCTION FDV$PUTDA : INTEGER; EXTERNAL;

[ASYNCHRONOUS] FUNCTION FDV$PUTL (
        val : [CLASS_S] PACKED ARRAY [$l1..$u1:INTEGER] OF CHAR) : INTEGER; EXTERNAL;

[ASYNCHRONOUS] FUNCTION FDV$PUTSC (
        fldnam : [CLASS_S] PACKED ARRAY [$l1..$u1:INTEGER] OF CHAR;
        fldval : [CLASS_S] PACKED ARRAY [$l2..$u2:INTEGER] OF CHAR := %IMMED 0) : INTEGER; EXTERNAL;

[ASYNCHRONOUS] FUNCTION FDV$READ (
        frmnam : [CLASS_S] PACKED ARRAY [$l1..$u1:INTEGER] OF CHAR;
        VAR memloc : [VOLATILE]ARRAY [$l2..$u2:INTEGER] OF INTEGER := %IMMED 0;
        size : INTEGER := %IMMED 0;
        VAR frmsiz : [VOLATILE]INTEGER := %IMMED 0) : INTEGER; EXTERNAL;

[ASYNCHRONOUS] FUNCTION FDV$RET (
        VAR fldval : [CLASS_S] PACKED ARRAY [$l1..$u1:INTEGER] OF CHAR;
        fldnam : [CLASS_S] PACKED ARRAY [$l2..$u2:INTEGER] OF CHAR;
        fldidx : INTEGER := %IMMED 0) : INTEGER; EXTERNAL;
```

```
[ASYNCHRONOUS] FUNCTION FDV$RETAL (
        VAR frmval : [CLASS_S] PACKED ARRAY [$11..$u1:INTEGER] OF CHAR) : INTEGER; EXTERNAL;

[ASYNCHRONOUS] FUNCTION FDV$RETCX (
        VAR tca : [VOLATILE]ARRAY [$11..$u1:INTEGER] OF INTEGER;
        VAR wksp : [VOLATILE]ARRAY [$12..$u2:INTEGER] OF INTEGER;
        VAR frmnam : [CLASS_S] PACKED ARRAY [$13..$u3:INTEGER] OF CHAR;
        VAR uarval : [CLASS_S] PACKED ARRAY [$14..$u4:INTEGER] OF CHAR;
        VAR curpos : [VOLATILE]INTEGER;
        VAR fldtrm : [VOLATILE]INTEGER;
        VAR insovr : [VOLATILE]INTEGER;
        VAR hlpnum : [VOLATILE]INTEGER) : INTEGER; EXTERNAL;

[ASYNCHRONOUS] FUNCTION FDV$RETDI (
        nmdidx : INTEGER;
        VAR nmdval : [CLASS_S] PACKED ARRAY [$12..$u2:INTEGER] OF CHAR;
        VAR nmdnam : [VOLATILE]INTEGER := %IMMED 0) : INTEGER; EXTERNAL;

[ASYNCHRONOUS] FUNCTION FDV$RETDN (
        nmdnam : [CLASS_S] PACKED ARRAY [$11..$u1:INTEGER] OF CHAR;
        VAR nmdval : [CLASS_S] PACKED ARRAY [$12..$u2:INTEGER] OF CHAR) : INTEGER; EXTERNAL;

[ASYNCHRONOUS] FUNCTION FDV$RETFN (
        VAR fldnam : [CLASS_S] PACKED ARRAY [$11..$u1:INTEGER] OF CHAR;
        VAR fldidx : [VOLATILE]INTEGER := %IMMED 0) : INTEGER; EXTERNAL;

[ASYNCHRONOUS] FUNCTION FDV$RETFO (
        fldnum : INTEGER;
        VAR fldnam : [CLASS_S] PACKED ARRAY [$12..$u2:INTEGER] OF CHAR;
        VAR fldidx : [VOLATILE]INTEGER) : INTEGER; EXTERNAL;

[ASYNCHRONOUS] FUNCTION FDV$RETFL (
        line : INTEGER;
        VAR val : [CLASS_S] PACKED ARRAY [$12..$u2:INTEGER] OF CHAR;
        VAR len : [VOLATILE]INTEGER := %IMMED 0) : INTEGER; EXTERNAL;

[ASYNCHRONOUS] FUNCTION FDV$RETLE (
        VAR fldlen : [VOLATILE]INTEGER;
        fldnam : [CLASS_S] PACKED ARRAY [$12..$u2:INTEGER] OF CHAR;
        fldidx : INTEGER := %IMMED 0) : INTEGER; EXTERNAL;

[ASYNCHRONOUS] FUNCTION FDV$RFRSH : INTEGER; EXTERNAL;
```

```
[ASYNCHRONOUS] FUNCTION FDV$SIGOP : INTEGER; EXTERNAL;

[ASYNCHRONOUS] FUNCTION FDV$SPADA (
        mode : INTEGER) : INTEGER; EXTERNAL;

[ASYNCHRONOUS] FUNCTION FDV$SPOFF : INTEGER; EXTERNAL;

[ASYNCHRONOUS] FUNCTION FDV$SPON : INTEGER; EXTERNAL;

[ASYNCHRONOUS] FUNCTION FDV$SSIGQ (
        sigmd : INTEGER) : INTEGER; EXTERNAL;

[ASYNCHRONOUS] FUNCTION FDV$SSRV (
        VAR status : [VOLATILE]INTEGER;
        VAR iostat : [VOLATILE]INTEGER := %IMMED 0) : INTEGER; EXTERNAL;

[ASYNCHRONOUS] FUNCTION FDV$STAT (
        VAR status : [VOLATILE]INTEGER;
        VAR iostat : [VOLATILE]INTEGER := %IMMED 0) : INTEGER; EXTERNAL;

[ASYNCHRONOUS] FUNCTION FDV$STERM (
        VAR tca : [VOLATILE]ARRAY [$l1..$ui:INTEGER] OF INTEGER) : INTEGER; EXTERNAL;

[ASYNCHRONOUS] FUNCTION FDV$STIME (
        time : INTEGER) : INTEGER; EXTERNAL;

[ASYNCHRONOUS] FUNCTION FDV$SWKSP (
        VAR wksp : [VOLATILE]ARRAY [$l1..$ui:INTEGER] OF INTEGER) : INTEGER; EXTERNAL;

[ASYNCHRONOUS] FUNCTION FDV$WAIT (
        VAR fldtrm : [VOLATILE]INTEGER := %IMMED 0) : INTEGER; EXTERNAL;

END.
```

Chapter 8
Programming FMS Applications in VAX–11 PL/I

The FMS Form Driver processes all Form Driver calls according to VAX–11 standards and language-specific rules. These rules define how arguments are passed to the Form Driver and how values are returned to your program. Language-specific information is briefly presented in this manual. For more detail, refer to the VAX–11 PL/I document set.

Your VAX–11 PL/I application program must comply with the requirements of the VAX–11 PL/I FMS interface. Topics discussed in this chapter include:

- Form Driver Routines
  - Invoking Form Driver Routines as Procedures
  - Accessing Form Driver Status Codes as Functions
- Argument Passing in FMS
- Null Arguments
- Defining the Entry Points
- FMS Data Types
  - Character Strings
  - Longword Binary Integers
  - Word Binary Integers
- Declarations
- Non-FMS Data Types
- One-Dimensional Arrays
- Allocation: Workspace, Terminal Control Area, and Run-Time Memory-Resident Form Area
- Precautions for Using FMS
- Data Conversion
- Sample Application Program in VAX–11 PL/I

A sample program written in PL/I (SAMPPLI.PLI) appears at the end of this chapter. Following the code for SAMPPLI.PLI are Form Driver definition files created for the sample program. Command file information needed to build the Sample Application program is in Section 8.12.2.

Examples from the Sample Application are used throughout the text to illustrate language issues. Where appropriate examples from SAMPPLI.PLI do not exist, other examples are provided.

8.1 Form Driver Routines

You can call any FMS routine as a subroutine or as a function. Syntax follows standard VAX–11 PL/I requirements.

8.1.1 Invoking Form Driver Routines as Procedures

You use the procedure call statement to invoke an FMS Form Driver routine. For example:

```
CALL FDV$WAIT;
```

Calls the Form Driver routine FDV$WAIT and passes no arguments.

```
CALL FDV$GET (OPTION, TERMINATOR, 'OPTION');
```

Calls the Form Driver routine FDV$GET and passes three arguments.

See Appendix A for a complete list of Form Driver calls. The calling sequence for each Form Driver routine, data access codes, data types, and passing mechanisms are presented in language-independent notation as specified by the VAX–11 Procedure Calling and Condition Handling Standard. For further detail about the VAX–11 Procedure Calling and Condition Handling Standard, refer to the *VAX–11 Run-Time Library Reference Manual.*

8.1.2 Accessing Form Driver Status Codes as Functions

An FMS status code is returned to the calling program at the completion of all Form Driver calls. To receive the returned status code from a Form Driver routine, you activate the routine with a function reference rather than with a call statement.

If you want to call an FMS routine as a function, you must specify the RETURNS option with a FIXED BIN (31) data type. The following statements reference FDV$GET as an FMS function:

```
DCL FDV$GET ENTRY (CHAR (*), FIXED BIN (31), CHAR (*), FIXED BIN (31)
    RETURNS (FIXED BIN (31)) OPTIONS (VARIABLE);
DCL STATUS FIXED BIN (31);
STATUS = FDV$GET (OPTION, TERMINATOR, 'OPTION');
```

8.2 Argument Passing in FMS

The argument passing mechanism refers to the way in which data is passed to a called routine. The VAX–11 Procedure Calling Standard has three methods for passing arguments:

- By reference
- By descriptor
- By value

FMS routines, however, expect arguments to be passed only by reference and by descriptor.

By reference specifies that the storage location of the argument is passed to the routine. FMS expects integers to be passed by reference, which is the PL/I default passing mechanism for integers.

By descriptor specifies that the address of a descriptor data structure is passed to the called routine. FMS expects character strings and arrays to be passed by descriptor. PL/I automatically passes character strings and arrays by descriptor if the parameter descriptors show a variable extent, as they do in the FMS include files FDVDEFCAL.PLI and FDVDEFFNC.PLI. Generally (in the EXTERNAL ENTRY declaration for FMS procedures), specify all character strings and integer arrays in the parameter list with (*). This will force PL/I to always pass arguments by descriptor, which is what FMS expects.

8.3 Null Arguments

When the call syntax includes optional arguments and you do not wish to specify all of the information, you can use null arguments. Any optional arguments can be omitted to simplify your program. A comma functions as a placeholder for each null argument. Optional arguments to the right of the last required argument can simply be omitted from the call. In the following example, the FDV$GETAL call passes only the field terminator value:

```
CALL FDV$GETAL (,FLDTRM);
```

8.4 Entry Point Definitions

The most difficult part of calling external routines from PL/I is defining the entry points. Every entry point used in a PL/I program must be declared with all its parameters and their types.

It is extremely important to have complete, correct definitions of all the entry points and their arguments. Your program may get warning messages if the number, data types, and uses of arguments in a call do not agree with their declarations. The include files FDVDEFCAL.PLI and FDVDEFFNC.PLI contain definitions for the entry points for the Form

Driver. FDVDEFCAL.PLI is the file for procedure calls. FDVDEFFNC.PLI is the file for function calls.

Many calls to FMS have a variable number of arguments. If you use the include file FDVDEFCAL.PLI or FDVDEFFNC.PLI, you do not have to worry about these variations in specified arguments because these include files have the entry points for the Form Driver defined.

8.5 FMS Data Types

8.5.1 Character Strings

The character string is one of the general data types used by FMS. For example, the FDV$GET call passes the character strings for field value (OPTION) and field name ('OPTION'):

```
CALL FDV$GET (OPTION, TERMINATOR, 'OPTION');
```

8.5.1.1 Defining Character Strings — Use only CHARACTER for strings passed to FMS. Do not use the CHARACTER VARYING attribute. You must be certain that your strings are declared to be long enough to accommodate your FMS data.

Two approaches are available for satisfying the fixed-length string requirements of FMS. One option is to declare your fixed-length strings to be the exact length of the FMS data to be returned. You can use the FMS/DESCRIPTION/DECLARATIONS command to get the length of the strings.

Alternatively, you can use a single string variable in different FMS calls to transfer data to or from several forms and fields. You must declare the string variable to be at least as large as the longest field value string that will be returned to your program. You can use the FDV$RETLE call to return the length of the valid portion of the field value in the string variable. This length can then be used when referencing the data that has been entered in the field.

```
DECLARE ACCOUNT CHARACTER(100);
  .
  .
  .
CALL FDV$GET (ACCOUNT, TERMINATOR, 'FIELD');
CALL FDV$RETLE (LENGTHFIELD, 'FIELD');
  .
  .
  .
PUT LIST (SUBSTR (ACCOUNT, 1, LENGTHFIELD));
```

After the execution of the FDV$RETLE call, LENGTHFIELD is equal to the length of the field named 'FIELD'. It is also equal to the valid portion of the variable ACCOUNT. LENGTHFIELD can now be used when referencing the data that was entered in the field named 'FIELD', and that is now in the variable ACCOUNT. If you do not use the PL/I SUBSTR function

when referencing ACCOUNT, you will reference the entire variable, including any blanks used by the Form Driver to pad the string.

A useful application of the FDV$RETLE call is in general purpose user action routines.

8.5.2 Longword Binary Integers

The longword binary integer is another general data type used by FMS. For example, the FDV$ATERM call passes the longword value for terminal control area size (12) and logical I/O channel number (2):

```
CALL FDV$ATERM (TCA, 12, 2);
```

Numeric arguments must be fixed binary (31) integers. If you try to pass other numeric types to the Form Driver, the calls will not work properly. An exception is the FDV$DFKBD call (see the following section).

8.5.3 Word Binary Integers

The defkbd argument is a word integer arrray passed when the FDV$DFKBD routine is called. FMS expects arrays to be passed by descriptor.

8.6 Declarations

If you do whole form processing with the FDV$GETAL, FDV$PUTAL, and FDV$RETAL routines, you can use the FMS/DESCRIPTION/DECLARATIONS command to produce a file of declarations describing the transferred data. Although these declarations are not directly usable in your PL/I program, they closely resemble PL/I syntax. You can edit them and include them in your application program.

8.7 Non-FMS Data Types

PL/I data types that are not recognized by FMS can be used in your PL/I application program provided they are not passed to the Form Driver. Using the Form Driver entry points correctly declared in your program, PL/I converts input arguments of non-FMS data type to arguments of the correct type by creating dummy arguments. However, you will not be given access to the values returned by the Form Driver to the output arguments. To allow non-FMS data types in your program to interact with FMS, use the PL/I conversion routines explicitly (see the VAX–11 PL/I document set).

8.8 One-Dimensional Arrays

One-dimensional arrays are structures that can be used in FMS for the following arguments:

- tca (terminal control area)
- wksp (workspace)
- mloc (memory location)
- defkbd (define keyboard)

You must provide FMS with storage space for these arguments. You can do that by defining them to be:

- contiguous longword integer arrays or character strings for tca, wksp, and mloc
- contiguous word integer arrays for defkbd

In the Sample Application program, the tca, wksp, and mloc arguments are passed to several Form Driver routines. These arguments are defined as integer array variables. You may alternatively define these arguments to be character strings. If you declare these variables to be character strings, you need to redefine all of the entry points that reference terminal control area, workspace, and memory location. Otherwise, you will get compile errors.

The following declarations establish names and storage for the integer array variables WORKSPACE, CHECKWKSP, TCA, and MENU_FORM:

```
DCL  WORKSPACE (3) FIXED BIN (31);   /* General workspace      */
DCL  CHECKWKSP (3) FIXED BIN (31);   /* Check workspace        */
DCL  TCA       (3) FIXED BIN (31);   /* Terminal Control Area */
DCL  MENU_FORM (500) FIXED BIN (31); /* Storage for memory-resident form*/
```

8.9 Allocation: Workspace, Terminal Control Area, and Run-Time Memory-Resident Form Area

The FMS workspace, terminal control area, and run-time memory-resident form area variables are character strings or longword integer arrays. These variables should be declared EXTERNAL. Note that this is not done in the Sample Application program. The sample program's structure protects the workspaces, terminal control areas, and run-time memory-resident form areas implicitly.

The Form Driver needs 12 bytes to associate user information about each workspace and terminal control area. The allocation for a run-time memory-resident form area must be the size of the form. The space for all of these variables is allocated by your application program. Note that FMS uses *only* 12 bytes of space for the workspace and terminal control area allocation. Any allocation greater than 12 bytes is wasted. You declare your workspace, terminal control area, and run-time memory-resident form area only once. FMS remembers their addresses after the addresses have been initially passed to the Form Driver.

For each workspace, the Form Driver also allocates an additional amount of storage based on your estimate of the amount of memory needed to store your largest form. If your estimate is too small, the Form Driver allocates more space automatically, but performance may be affected. An adequate estimate results in a more efficient operation of the Form Driver. You can use the FMS/DIRECTORY/FULL command to find out how much space to allocate.

In the following example from the Sample Application program, workspace is allocated and the FDV$AWKSP call is issued. When the FDV$AWKSP routine is called, the first argument (WORKSPACE) specifies the area of memory to be used for your workspace. The second argument specifies an estimate of the workspace size (2000 bytes) that you will need to display the largest form in your application.

```
DCL WORKSPACE (3) FIXED BIN (31);   /* General workspace */
CALL FDV$AWKSP (WORKSPACE, 2000);
```

8.10 Precautions for Using FMS

8.10.1 Memory Areas Used Exclusively by FMS

The locations for terminal control area, workspace, and run-time memory-resident form area are used exclusively by FMS. The terminal control area and workspace are attached with the FDV$ATERM and FDV$AWKSP calls and remain allocated until the FDV$DTERM and FDV$DWKSP calls are issued or until the program ends. The run-time memory-resident form area, used in the FDV$READ call, remains allocated until the FDV$DEL call is issued or until the program ends. You never touch the terminal control areas, workspaces, or run-time memory-resident form areas in your program except to pass their addresses to the Form Driver.

8.10.2 Why You Should Use the EXTERNAL Attribute

Parameters to the following Form Driver routines should be used with caution:

FDV$ATERM	Attach terminal
FDV$AWKSP	Attach form workspace
FDV$READ	Read form into memory
FDV$SSRV	Specify status reporting variables

For example, once an FDV$SSRV call is issued, the program variables that contain the FMS status and RMS status become volatile and can change at any call point. As a general rule, you should place the status reporting variables in static storage.

In cases where you need both the FMS and RMS statuses, the FDV$STAT routine can be used. Note that only the FDV$STAT and FDV$SSRV calls provide RMS status. With the FDV$STAT routine, you do not have to worry about volatility.

The locations for terminal control area, workspace, run-time memory-resident form area, and status reporting variables must all continue to exist while the Form Driver is using them. They must remain allocated until the terminal control area and workspace are detached, until forms in memory location are deleted, and until the status reporting variables are no longer

used. The variables can be protected by declaring them EXTERNAL; otherwise, the compiler might place them in dynamic storage or reuse their storage area.

8.11 Data Conversion

FMS uses only ASCII character strings to display data. All information displayed on the terminal screen, and all information received from the terminal operator, is represented as ASCII field values. Any manipulation of numeric data requires conversion of ASCII character strings to numeric data, and conversion of numeric data back to ASCII character strings. In the following discussion of conversion routines, you should assume that the receiving data type can support the largest number that is likely to be generated.

In the Sample Application, the following steps are taken to get a new account balance after writing a check:

```
CALL FDV$RET (REGARRAY.AMTPAY (LASTREGNUM), 'AMTPAY');
AMTPAY = FIXED (REGARRAY.AMTPAY (LASTREGNUM), 31);
BALANCE = BALANCE - AMTPAY;
TOTPAY = TOTPAY + AMTPAY;

CALL FDV$PUT (CENTS (BALANCE), 'BALANCE');

CENTS: PROCEDURE (PENNIES) RETURNS (CHAR(*));
DCL PENNIES              FIXED BIN (31);
DCL CHAR_PENNIES         PICTURE 'ZZZ999';

CHAR_PENNIES = PENNIES;
RETURN (CHAR_PENNIES);

END CENTS;
```

In this example, the PL/I FIXED function is used to convert the string expression REGARRAY.AMTPAY(LASTREGNUM) to a fixed-point integer variable with a specified number of binary digits (31) used to hold the data item's value. The integer value of AMTPAY is subtracted from the integer value of BALANCE to produce a new value for BALANCE. The value of AMTPAY is also added to the integer value of TOTPAY to produce a new value for TOTPAY.

After the data operations have been completed, an integer-to-ASCII character string conversion is accomplished. Using an assignment to a picture variable, the user-created CENTS function is used to convert the integer variable BALANCE to an ASCII character expression BALANCE. The value for the balance is displayed in a right-justified field 'BALANCE'. The rightmost digit from the program is displayed in the field's rightmost character position. The remaining digits of the character expression are placed to the left of the rightmost digit. If input is longer than the field, FMS truncates on the left. (The Form Driver displays a data length error message (FDV$_DLN) only if you have set FMS Debug mode.)

Note that in this example the output goes to a field with a decimal point field-marker character. In the presence of a decimal point field marker, the Form Driver creates strange-looking output for single-digit data items. The

output will be a period followed by a space and then digit — rather than .01, for example. In the above example, the assignment to a picture variable is used to prevent this kind of unconventional output.

The PL/I built-in function CHARACTER converts a number to a character string with one or two leading blanks (see the VAX–11 PL/I documentation for details). If you display this string in a left-justified field, you must take these leading blanks into consideration. Your program can use the CHARACTER function for data conversion operations if field markers will not create a confusing appearance.

For other conversion options, see the general conversion routines in the *VAX–11 Run-Time Library Reference Manual.*

8.12 Sample Application Program in VAX–11 PL/I

The FMS Sample Application program (SAMPPLI.PLI) is part of the FMS distribution kit. When FMS is installed, SAMPPLI.PLI is placed in the directory FMS$EXAMPLES. Designed to be a demonstration program and learning tool, SAMPPLI.PLI shows most of the features provided by FMS. The entire Sample Application program appears at the end of this chapter.

8.12.1 Form Driver Definition Files

The files FDVDEFCAL.PLI and FDVDEFFNC.PLI are part of the Sample Application program package. When FMS is installed, FDVDEFCAL.PLI and FDVDEFFNC.PLI are placed in the directory FMS$EXAMPLES. The FDVDEFCAL.PLI and FDVDEFFNC.PLI files appear after the Sample Application source code.

FDVDEFCAL.PLI and FDVDEFFNC.PLI contain a variety of codes for the Form Driver routines used in the Sample Application program. Although these codes have been created for use in SAMPPLI.PLI, they can provide you with a helpful starting point as you create definitions for your own application program.

FDVDEFCAL.PLI is the include file for SAMPPLI.PLI with all Form Driver references as calls. FDVDEFFNC.PLI is the include file for SAMPPLI.PLI with all Form Driver references as functions. The files FDVDEFCAL.PLI and FDVDEFFNC.PLI include:

- FMS terminator codes
- Function key terminators returned from the FDV$GET and FDV$WAIT calls
- Form Driver key functions for use with the FDV$DFKBD call
- User action routine (UAR) return codes, which are returned by the UARs to the Form Driver:

  Field completion UAR return codes

  Help UAR return codes

  Function key UAR return codes

- VMS status codes returned when Form Driver routines are called as functions. These codes can be signaled.
- FMS status codes returned when the FDV$STAT routine is called as a function
- Declarations of Form Driver routines

8.12.2 Command File for Building the Sample Application Program

The command file for building the Sample Application program includes all the information that you need to compile and link SAMPPLI.PLI. When FMS is installed, the command file is placed in the directory FMS$EXAMPLES.

```
$!     S A M P P L I . C O M
$!
$!     Compile and link the PL/I version of the FMS V2 Sample Application
$!
$!     The PLI source files are:          SAMPPLI.PLI
$!                                        FDVDEFCAL.PLI
$!
$!     SMPVECTOR.OBJ and SMPMEMRES.OBJ were produced by the FMS commands:
$!
$!       $ FMS/VECTOR/OUTPUT=SMPVECTOR    SAMP.FLB
$!       $ FMS/MEMORY/OUTPUT=SMPMEMRES SAMP.FLB/FORM=(HELP_KEYS,HELP_MENU)
$!
$ PLI    SAMPPLI
$ LINK   SAMPPLI, FMS$EXAMPLES:SMPVECTOR, FMS$EXAMPLES:SMPMEMRES
```

```
/*************************************************/
/*                                               */
/* SAMP -- The FMS V2 Sample Application Program */
/*                                               */
/*************************************************/

SAMP : PROCEDURE OPTIONS(MAIN);

/* Data definitions
/*
/* FMS related
/*                                                              */
DCL WORKSPACE( 3 ) FIXED BIN(31);       /* General workspace    */
DCL CHECKWKSP( 3 ) FIXED BIN(31);       /* Check workspace      */
DCL TCA( 3 )       FIXED BIN(31);       /* Terminal Control Area*/
/*
/* Storage for forms read onto memory resident list by FDV$READ
/*                                                              */
DCL MENU_FORM(  500) FIXED BIN(31),
    CHECK_FORM( 750) FIXED BIN(31),
    DPOSIT_FORM(500) FIXED BIN(31);
/*                              */
/* Account (Read in from file)  */
/*                              */
DCL 1 ACCOUNT STATIC,
        2 ACCTNO        CHAR(5),
        2 ACCDATE       CHAR(7),
        2 LAST          CHAR(20),
        2 FIRST         CHAR(15),
        2 MIDDLE        CHAR(15),
        2 STREET        CHAR(30),
        2 CITY          CHAR(20),
        2 STATE         CHAR(2),
        2 ZIP           CHAR(5),
        2 HOMEPH        CHAR(10),
        2 WORKPH        CHAR(10),
        2 OPW           CHAR(12);
DCL ACCOUNT_STR         CHAR(151) DEFINED ( ACCOUNT );
DCL 1 TACCOUNT STATIC,
        2 TEMPACCOUNT   CHAR(151);
```

```
/*                                              */
/* Deposit data (Read via FDV$GETAL)            */
/*                                              */
DCL 1 DEPOSIT STATIC,
        2 DATE          CHAR(7),
        2 CURBAL        CHAR(6),
        2 AMT           CHAR(6),
        2 NEWBAL        CHAR(6),
        2 MEMO          CHAR(35);
DCL DEPOSIT_STR         CHAR(60) DEFINED ( DEPOSIT );

/* Money.
/* Note that all money is kept internally as integers (in cents).
/* It is only when the quantities are output that they look like
/* dollars, since all the money fields have periods as field
/* markers in the right places and they are right justified or
/* fixed decimal.                                                  */
/*                                                                 */
/* Register Data                                                   */
/* Keep storage only for 30 lines of register data.                */
%REPLACE REGSIZE BY 30;
DCL 1 REGARRAY( REGSIZE ) STATIC,
        2 NUM           CHAR(4),        /*Check number if a check, blank else*/
        2 DATE          CHAR(7),        /*Date of action*/
        2 MEMPAYTO      CHAR(35),       /*Payto if check, memo if deposit*/
        2 AMTDEP        CHAR(6),        /*Amt deposited, if deposit*/
        2 AMTPAY        CHAR(6),        /*Amt paid, if a check*/
        2 BALANCE       CHAR(6);        /*Balance after action*/
DCL REGARRAY_STR( 30 )  CHAR(64) DEFINED( REGARRAY );
```

```
/* Other variables */
DCL     TERMINATOR          FIXED BIN,      /*Terminator returned by FDV*/
        BALANCE             FIXED BIN,      /*Balance in account, numeric*/
        SBALANCE            FIXED BIN,      /*Starting balance */
        AMTPAY              FIXED BIN,      /*Check payment amount*/
        TOTDEP              FIXED BIN,      /*Total deposits made in this session*/
        TOTPAY              FIXED BIN,      /*Total checks payed in this session*/
        FMSSTATUS           FIXED BIN,      /*Status for last FDV call*/
        RMSSTATUS           FIXED BIN,      /*RMS Status for last FDV call*/
        LASTREGNUM          FIXED BIN,      /*Last number used in the register (1...REGSIZE)*/
        LASTCHNUM           FIXED BIN,      /*Last check number used*/
        PASSWORD            CHAR(12),       /*Password from account*/
        JUNK                CHAR(100),      /*Temporary storage for return from GETAL*/
        SIZE_MENU           FIXED BIN,      /*Gets size of menu form*/
        SIZE_CHECK          FIXED BIN,      /*Gets size of check form*/
        SIZE_DPOSIT         FIXED BIN,      /*Gets size of dposit form*/
        CURLINE             FIXED BIN,      /*Line of check register that cursor is now on*/
        MINWINDOW           FIXED BIN,      /*Smallest line of register being displayed
                                            on the scrolled area*/
        MAXWINDOW           FIXED BIN;      /*Largest line of register beig displayed
                                            on the scrolled area*/

DCL TRM$ EXTERNAL ENTRY( CHAR(*) ) RETURNS( CHAR(*) );
                                            /*Returns string with no trailing blanks*/

%INCLUDE 'FDVDEFCAL.PLI';
```

```
/* MAIN ROUTINE OF SAMP
/*
/* Initialize FMS
/*   Attach default terminal to channel 2
/*   Attach normal and check workspaces (order important for help
/*      and refresh during CHECK/CHECKDONE time--try switching and see).
/*   Open form library, attach to channel 1
/*   Set keypad mode to application
/*   Set signal mode to bell (default, but it's fun to do)
/*                                                                    */
CALL FDV$ATERM( TCA, 12, 2 );                    CALL GETSTA;
CALL FDV$AWKSP( CHECKWKSP, 2000 );               CALL GETSTA;
CALL FDV$AWKSP( WORKSPACE, 2000 );               CALL GETSTA;
CALL FDV$LOPEN( 'FMS$EXAMPLES:SAMP', 1 );        CALL GETSTA;
CALL FDV$SPADA( 1 );
CALL FDV$SSIGQ( 0 );
/*
/* Set all future calls to return status to the two status recording
/* variables FMSSTATUS and RMSSTATUS without having to call the
/* the FDV$STAT routine.
/*                                                                    */
CALL FDV$SSRV( FMSSTATUS, RMSSTATUS );
/*
/* Read in a few forms from the form library onto the dynamic
/* resident form list.  You may be able to detect the difference
/* in the form to form access times for those forms which have to be
/* accessed from the form library on disk and those forms which are
/* on the dynamic or static memory resident form list.  See the
/* installation notes for this program (the LINK command) to see
/* which forms are on the static memory resident form list.
/*                                                                    */
CALL FDV$READ( 'MENU',  MENU_FORM,   2000, SIZE_MENU );
CALL FDV$READ( 'CHECK', CHECK_FORM,  3000, SIZE_CHECK );
CALL FDV$READ( 'DEPOSIT',DPOSIT_FORM, 2000, SIZE_DPOSIT );
/*
/* Initialize account information
/*                                                                    */
CALL INACCT;
```

```
/*
/* Put up welcome form, wait for response
/*                                                                    */
CALL FDV$CDISP( 'WELCOME' );      CALL SRVCHK;
CALL FDV$WAIT;
/*
/* Process all menu requests
/*                                                                    */
CALL MENU;
/*
/* Clean up and leave:
/*   Close form library.
/*   Reset Keypad to numeric.
/*   Delete a form from dynamic memory resident form list just to show how.
/*   Detach workspaces (not really necessary since DTERM would do it).
/*   Detach terminal.
/*                                                                    */
CALL FDV$LCLOS;
CALL FDV$SPADA( 0 );
CALL FDV$DEL( 'MENU' );
CALL FDV$DWKSP( WORKSPACE );
CALL FDV$DWKSP( CHECKWKSP );
CALL FDV$DTERM( TCA );
```

```
INACCT: PROCEDURE;
/*****************************************************************************/
/* Subroutine INACCT
/*      Read from file SAMP.DAT into internal variables.
/*      Set up the workspace for checks and fill in the check form
/*      with the account's name, address, and account number.
/*****************************************************************************/
/*
/* Open file, set account data
/*                                                                          */
DCL ACCOUNTFILE STREAM INPUT;
DCL EOF BIT INITIAL('0'B);

ON ENDFILE( ACCOUNTFILE )
   BEGIN;
      CLOSE FILE( ACCOUNTFILE );
      EOF = '1'B;
   END;

OPEN FILE( ACCOUNTFILE ) TITLE( 'FMS$EXAMPLES:SAMP.DAT' );
GET FILE( ACCOUNTFILE ) EDIT ( ACCOUNT_STR ) ( A( 151 ) );
/*
/* Read the remaining records into the check register, counting them.
/* The last record has the current balance, and some record has the
/* last check number used (not necessarily the last record).
/*                                                                          */
LASTREGNUM = 0;
LASTCHNUM = 0;
GET FILE( ACCOUNTFILE ) EDIT( REGARRAY_STR( 1 )) (A(64));
DO WHILE (LASTREGNUM < REGSIZE   EOF);
    LASTREGNUM = LASTREGNUM + 1;
    IF REGARRAY.NUM( LASTREGNUM ) = '    '
    THEN LASTCHNUM = FIXED( REGARRAY.NUM( LASTREGNUM ), 31 );
    IF LASTREGNUM < REGSIZE
    THEN GET FILE( ACCOUNTFILE ) EDIT( REGARRAY_STR( LASTREGNUM + 1 )) (A(64));
END;
/*
/* Reach here as result of end of file or filled register.
/* Check for data file in error.
/* Take balance from last record read.
/* Set session sums to zero to say no activity yet.
/*                                                                          */
```

```
IF LASTREGNUM = 0
THEN DO;
        PUT SKIP LIST( 'DATA FILE IN ERROR' );
        STOP;
     END;
BALANCE = FIXED( REGARRAY.BALANCE( LASTREGNUM ), 31 );
SBALANCE = BALANCE;
TOTDEP = 0;
TOTPAY = 0;
/*
/* Set up the check workspace once so we don't have to do it every time.
/*                                                                    */
CALL FMTCHK;

END INACCT;



FMTCHK: PROCEDURE;
/*************************************************************************/
/* Subroutine FMTCHK
/*      Format account data onto check form in the check workspace.
/*************************************************************************/

CALL FDV$SWKSP( CHECKWKSP );
CALL FDV$LOAD( 'CHECK' );
CALL FDV$PUT( TRIM( FIRST ) || ' ' || SUBSTR( MIDDLE, 1, 1 ) || '. '
           || TRIM( LAST ), 'NAME' );
CALL FDV$PUT( STREET, 'STREET' );
CALL FDV$PUT( TRIM( CITY ) || ', ' || STATE || ' ' || ZIP , 'CSZ' );
CALL FDV$PUT( HOMEPH, 'HOMEPH' );
CALL FDV$PUT( ACCTNO, 'ACCTNO' );
CALL FDV$SWKSP( WORKSPACE );
END FMTCHK;
```

```
MENU: PROCEDURE;
/*****************************************************************************/
/* Subroutine MENU
/*      Accept inputs from the menu form and dispatch to the
/*      appropriate routine.  Repeat until option 1 (exit) is
/*      chosen.  The UARs in the form guarantee that we get back
/*      only inputs '1'-'5' with the correct terminators.
/*      Options are:
/*              1 => Exit
/*              2 => Write checks
/*              3 => Make deposit
/*              4 => View register
/*              5 => View account data
/*****************************************************************************/

DCL OPTION CHAR(1) INIT( '' );  /*Choice returned from menu*/

DO WHILE(OPTION = '1');
    CALL FDV$CDISP( 'MENU' );    CALL SRVCHK;
    CALL FDV$GET( OPTION, TERMINATOR, 'OPTION' );
    IF OPTION = '1' THEN RETURN;
    ELSE IF OPTION = '2' THEN CALL WRITCH;
    ELSE IF OPTION = '3' THEN CALL MAKDEP;
    ELSE IF OPTION = '4' THEN CALL VUEREG;
    ELSE IF OPTION = '5' THEN CALL VUEACT;
END;
END MENU;
/*              */



WRITCH: PROCEDURE;
/*****************************************************************************/
/* Subroutine WRITCH
/*      Write one or more checks
/*****************************************************************************/
/*
/* Turn on LED 3 on the VT100 during this routine, just to show how.
/*                                                                  */
CALL FDV$LEDON( 3 );
```

```
/*
/* Maark WORKSPACE not displayed so it doesn't show up during a refresh.
/* Put up CHECK form from already loaded workspace
/* and display current balance
/*                                                                    */
CALL FDV$NDISP;
CALL FDV$SWKSP( CHECKWKSP );
CALL FDV$DISPW;
CALL FDV$PUT( CENTS( BALANCE ), 'BALANCE' );
/*
/* Process checks until a Keypad period is read
/*                                                                    */
TERMINATOR = 0;
DO WHILE(TERMINATOR = FDV$K_KP_PER);
    CALL ONECHK;                          /* Process one check */
    CALL ENDCHK;                          /* Give options for continuing*/
END;
/*
/* Turn off LED 3 on VT100
/*                                                                    */
CALL FDV$LEDOF( 3 );
CALL FDV$SWKSP( WORKSPACE );
END WRITCH;



ONECHK: PROCEDURE;
/*************************************************************************/
/* Subroutine ONECHK -- Process one check
/*      If input is terminated by Kpd period, return with no action
/*      Else deduct from balance and enter into register.
/*      Note that a UAR in the form guarantees that the amount of
/*      the check is always less than or equal to the balance.
/*      Note that the form function key UAR  allows only Kpd period
/*      as terminator (other than FDV$K_FT_NTR).
/*************************************************************************/

CALL FDV$PUT( CHARACTER( LASTCHNUM + 1 ), 'NUMBER' );
CALL FDV$GETAL( JUNK, TERMINATOR );
IF TERMINATOR = FDV$K_KP_PER THEN RETURN;
```

```
/*
/* If the check wouldn't fit in the register, don't process, just
/* give error message, wait for acknowledgement, and return
/*                                                                  */
IF LASTREGNUM = REGSIZE
THEN DO;
        CALL FDV$PUTL( 'Register full, can''t enter check' );
        CALL FDV$WAIT;
        RETURN;
     END;
/*
/* Update register and counters
/*                                                                  */
LASTREGNUM = LASTREGNUM + 1;
LASTCHNUM = LASTCHNUM + 1;

/*
/* Get amount from check.
/* Update balance (in memory and on screen) and session sums.
/* Transfer form values to register item.
/*                                                                  */
CALL FDV$RET( REGARRAY.AMTPAY(LASTREGNUM), 'AMTPAY' );
AMTPAY = FIXED( REGARRAY.AMTPAY(LASTREGNUM), 31 );
BALANCE = BALANCE - AMTPAY;
TOTPAY = TOTPAY + AMTPAY;

CALL FDV$PUT( CENTS( BALANCE ), 'BALANCE' );
CALL FDV$RET( REGARRAY.BALANCE(LASTREGNUM), 'BALANCE'); /*Avoids need to format RI.BALANCE*/

REGARRAY.AMTDEP(LASTREGNUM) = '';
CALL FDV$RET( REGARRAY.NUM(LASTREGNUM), 'NUMBER' );
CALL FDV$RET( REGARRAY.DATE(LASTREGNUM), 'DATE' );
CALL FDV$RET( REGARRAY.MEMPAYTO(LASTREGNUM), 'PAYTO' ); /*Note: not from check's MEMO*/

END ONECHK;
```

```
ENDCHK: PROCEDURE;
/*****************************************************************************/
/* Subroutine ENDCHK
/*      Finish off check processing by giving operator
/*      three options:
/*              RETURN  Write another check
/*              KPD 0   Print the check into file SAMPCH.DAT
/*              KPD .   Return to menu
/*      Check to see if check write was aborted by Kpd per.
/*      If so, then don't give any further choice, just abort.
/*      Note that form function key UAR allows only the above
/*      terminators to get through.
/*****************************************************************************/
IF TERMINATOR = FDV$K_KP_PER THEN RETURN;
/*
/* Tell the operator that the check has been paid by overlaying with
/* a new form, using the normal workspace, thereby saving the check
/* workspace in case another check is to be written.
/*                                                                   */
CALL FDV$SWKSP( WORKSPACE );
CALL FDV$DISP( 'CHECK_DONE' );   CALL SRVCHK;
/*
/* Wait for operator to enter either KPD period, NTR, or KPD zero.
/* Print the check as many times as requested.
/* (Note that a UAR on the form guarantees that only those terminators
/* are accepted).
/* Process accordingly.
/*                                                                   */
CALL FDV$WAIT( TERMINATOR );
DO WHILE (TERMINATOR = FDV$K_KP_0);
    CALL PRICHK;                    /* Print the check */
    CALL FDV$WAIT( TERMINATOR );
END;
/*
/* If choice is to quit,
/* then mark check wksp not displayed so it doesn't appear during refresh
/* else mark normal workspace (occupied by CHECKDONE form) not displayed
/*   so it doesn't show during refresh and then clear its lines.
/*   (Clearing the space occupied by the CHECKDONE form, lines 20-23,
/*   is beter done by overlaying with a blank form to
/*   avoid having to know the line numbers to clear).
/*                                                                   */
```

```
IF TERMINATOR = FDV$K_KP_PER
THEN DO;
        CALL FDV$SWKSP( CHECKWKSP );
        CALL FDV$NDISP;
     END;
ELSE DO;
        CALL FDV$NDISP;
        CALL FDV$CLEAR( 20, 4 );
        CALL FDV$SWKSP( CHECKWKSP );
     END;

/*
/* Going to write another check now or eventually, so:
/* Clear out operator entered fields.
/*                                                                    */
CALL FDV$PUTD( 'AMTPAY' );
CALL FDV$PUTD( 'MEMO' );
CALL FDV$PUTD( 'PAYTO' );
END ENDCHK;



PRICHK: PROCEDURE;
/*************************************************************************/
/* Subroutine PRICHK
/*      Print the check into the file SAMPCH.DAT
/*      Use the check workspace, then switch back to the normal wksp
/*      to keep things clean.
/*************************************************************************/
DCL     SYSPRINT        FILE PRINT,
        FIRSTL          CHAR(80),       /*First line on the form of the check
                                        image (from named data)*/
        LASTL           CHAR(80),       /*Last line on the form of the check
                                        image (from named data)*/
        LINE            CHAR(80),       /*Line return as image of form for check print*/
        LINE_LENGTH     FIXED BIN,      /*Length of line image returned*/
        I               FIXED BIN;      /*Index into lines of check*/
/*
/* Open check writing file.  Note there's a new version for every check.
/* Switch workspaces
/*                                                                    */
```

```
OPEN FILE(SYSPRINT) TITLE('SAMPCH.DAT');
CALL FDV$SWKSP( CHECKWKSP );
/*
/* Get the top and bottom lines of the check from the named data
/* (first two characters).
/*                                                                    */
CALL FDV$RETDN( 'FIRST', FIRSTL );        CALL SRVCHK;
CALL FDV$RETDN( 'LAST', LASTL );          CALL SRVCHK;
/*
/* Get lines from form.
/* Write to file.
/*                                                                    */
DO I = FIXED( FIRSTL, 31 ) TO FIXED( LASTL, 31 );
    CALL FDV$RETFL( I, LINE, LINE_LENGTH );
    PUT FILE( SYSPRINT ) SKIP LIST( SUBSTR( LINE, 1, LINE_LENGTH ) );
END;
CALL FDV$PUTL( 'Check written to file' );
CLOSE FILE( SYSPRINT );
CALL FDV$SWKSP( WORKSPACE );
END PRICHK;



MAKDEP: PROCEDURE;
/*************************************************************************/
/* Subroutine MAKDEP
/*      Make a deposit, enter into check register
/*      Cancel on Keypad period.
/*      Note that the form function key UAR allows only Kpd period.
/*************************************************************************/
DCL     DONE            CHAR(80);       /*Form done message for Deposit*/

/*
/* Put up deposit form with current balance
/*                                                                    */
CALL FDV$CDISP( 'DEPOSIT' );     CALL SRVCHK;
CALL FDV$PUT( CENTS( BALANCE ), 'CURBAL' );
/*
/* Get deposit amount and memo from operator.
/* Abort on Kpd period.
/*                                                                    */
```

```
CALL FDV$GETAL( DEPOSIT_STR, TERMINATOR );
IF TERMINATOR = FDV$K_KP_PER THEN RETURN;
/*
/* Have deposit information now. If no room in check register
/* must abort.
/*                                                                    */
IF LASTREGNUM = REGSIZE
THEN DO;
        CALL FDV$PUTL( 'Register full, can''t enter deposit' );
        CALL FDV$WAIT;
        RETURN;
     END;
/*
/* Add to balance and session sum.
/* Check for overflow (program and form keep only six digits).
/* Display new balance.
/* Make entry in register.
/*                                                                    */
BALANCE = BALANCE + FIXED( DEPOSIT.AMT, 31 );
TOTDEP = TOTDEP + FIXED( DEPOSIT.AMT, 31 );
IF BALANCE >= 1000000
THEN DO;
        BALANCE = BALANCE - 1000000;
        CALL FDV$PUTL( 'Overflow in bank computer, only 6 digits kept' );
        CALL FDV$WAIT;
     END;
CALL FDV$PUT( CENTS( BALANCE ), 'NEWBAL' );
LASTREGNUM = LASTREGNUM + 1;
REGARRAY.NUM(       LASTREGNUM ) = ' ';   /* Blank since it's not a check*/
REGARRAY.DATE(      LASTREGNUM ) = DEPOSIT.DATE;
REGARRAY.MEMPAYTO( LASTREGNUM ) = DEPOSIT.MEMO;
REGARRAY.AMTDEP(    LASTREGNUM ) = DEPOSIT.AMT;
REGARRAY.AMTPAY(    LASTREGNUM ) = '';
CALL FDV$RET( REGARRAY.BALANCE( LASTREGNUM ), 'NEWBAL' );
                        /* Avoids need to format REGARRAY.BALANCE(LASTREGNUM)*/
/*
/* Sample of how to keep message texts stored with the form rather
/* than in a program.  This is especially useful for multi-lingual
/* environments; only the form text and the form named data must
/* be changed and nothing in the program.  The trick is to store the
/* response text in named data. This is the only example of how to do
```

```
/* it in this program, but all messages could be stored like this.
/* Message intent is: "Deposit made, press RETURN or ENTER to continue."
/*                                                                  */
CALL FDV$RETDN( 'DONE', DONE );
CALL FDV$PUTL( DONE );
CALL FDV$WAIT;
END MAKDEP;



VUEREG: PROCEDURE;
/*************************************************************************/
/* Subroutine VUEREG
/*      View the check register and scroll throush it.
/*      Also display totals for current session.
/*************************************************************************/

DCL     NSCROLC         CHAR(2),        /*From named data,   line in scr area*/
        NSCROL          FIXED BIN,      /*Integer version of " */
        FAKE            CHAR(1);        /*Value returned from fake field in scrolled area*/

/*
/* Put up register form.
/* Check for current session totals overflow. If so, output 'OVRFLO'
/* Put out summary of this session into indexed(4) fields.
/*                                                                  */

CALL FDV$CDISP( 'REGISTER' );    CALL SRVCHK;
CALL FDV$PUT( CENTS( SBALANCE ),        'SUMMARY', 1 );
IF TOTDEP < 1000000
THEN CALL FDV$PUT( CENTS( TOTDEP ),     'SUMMARY', 2 );
ELSE CALL FDV$PUT( 'OVRFLO',            'SUMMARY', 2 );
IF TOTPAY < 1000000
THEN CALL FDV$PUT( CENTS( TOTPAY ),     'SUMMARY', 3 );
ELSE CALL FDV$PUT( 'OVRFLO',            'SUMMARY', 3 );
CALL FDV$PUT(      CENTS( BALANCE ),    'SUMMARY', 4 );
/*
/* Get number of lines in scroll area from form named data (item 1).
/*                                                                  */
CALL FDV$RETDI( 1, NSCROLC );            CALL SRVCHK;
NSCROL = FIXED( NSCROLC, 31 );
```

```
/*
/* Put lines from check register array into scrolled area.
/* The window is initially from item 1 up to item
/* min(NSCROL,LASTREGNUM), that is, up to the size of the scrolled
/* area or the size of the register, whichever is less. Assume there
/* is at least one line (the initial deposit).
/*                                                              */
MINWINDOW = 1;
CALL FDV$PUTSC( 'NUMBER', REGARRAY_STR(1) );     /* First line*/
CURLINE = 1;                                     /* Reg item cursor is on*/
DO WHILE ( CURLINE < LASTREGNUM  CURLINE < NSCROL );
    CURLINE = CURLINE + 1;
    CALL FDV$PFT( FDV$K_FT_SFW, 'NUMBER' );
    CALL FDV$PUTSC( 'NUMBER', REGARRAY_STR( CURLINE ) );
END;
MAXWINDOW = CURLINE;
/*
/* Get input from fake field of scrolled line and do what it says:
/*      Kpd . or RETURN/ENTER    => return to menu
/*      UPARROW or TAB           => scroll forward
/*      DOWNARROW or BACKSPACE   => scroll backward
/*      all others               => ignore
/* Note that there is no form function key UAR so this routine
/* handles all terminators itself (by ignoring illegal ones).
/*                                                              */
CALL FDV$GET( FAKE, TERMINATOR, 'FAKE' );
DO WHILE (( TERMINATOR = FDV$K_FT_NTR | TERMINATOR = FDV$K_KP_PER ));
    IF TERMINATOR = FDV$K_FT_SFW | TERMINATOR = FDV$K_FT_SNX THEN CALL SCRFWD;
    IF TERMINATOR = FDV$K_FT_SBK | TERMINATOR = FDV$K_FT_SPR THEN CALL SCRBAK;
    CALL FDV$GET( FAKE, TERMINATOR, 'FAKE' );
END;
END VUEREG;
```

```
SCRFWD: PROCEDURE;
/*****************************************************************************/
/* Subroutine SCRFWD -- Scroll forward.
/*      CURLINE is the line in the resister that the cursor is on.
/*      MINWINDOW and MAXWINDOW delimit the part of the resister
/*      currently displayed in the scrolled area
/*****************************************************************************/
/*
/* If cursor is at the end of the resister, report, and return
/*                                                                           */
IF CURLINE = LASTREGNUM
THEN DO;
        CALL FDV$PUTL( 'Last line of resister' );
        RETURN;
     END;
/*
/* If cursor not at the last line of a window, just move down
/* If cursor is  at the last line of a window,
/*      move window forward one line,
/*      write the new last line to the last line of the scrolled area
/* Move current line pointer forward
/*                                                                           */
IF CURLINE = MAXWINDOW
THEN CALL FDV$PFT( FDV$K_FT_SFW, 'NUMBER' );
ELSE DO;
        MINWINDOW = MINWINDOW + 1;
        MAXWINDOW = MAXWINDOW + 1;
        CALL FDV$PFT( FDV$K_FT_SFW, 'NUMBER', REGARRAY_STR( MAXWINDOW ) );
     END;
CURLINE = CURLINE + 1;
END SCRFWD;
```

```
SCRBAK: PROCEDURE;
/*****************************************************************************/
/* Subroutine SCRBAK -- Scroll backward
/*      CURLINE is the line in the register that the cursor is on.
/*      MINWINDOW and MAXWINDOW delimit the part of the register
/*      currently displayed in the scrolled area
/*****************************************************************************/
/*
/* If the cursor is at the beginning of the register, report, and return
/*                                                                          */
IF CURLINE = 1
THEN DO;
        CALL FDV$PUTL( 'First line of register' );
        RETURN;
     END;
/*
/* If cursor not at first line of the window, just move up
/* If cursor is  at first line of the window,
/*      move window back one line,
/*      write the new first line to the first line of the scrolled area
/* Move current line pointer back
/*                                                                          */
IF CURLINE = MINWINDOW
THEN CALL FDV$PFT( FDV$K_FT_SBK, 'NUMBER' );
ELSE DO;
        MINWINDOW = MINWINDOW - 1;
        MAXWINDOW = MAXWINDOW - 1;
        CALL FDV$PFT( FDV$K_FT_SBK, 'NUMBER', REGARRAY_STR( MINWINDOW ) );
     END;
CURLINE = CURLINE - 1;
END SCRBAK;
```

```
VUEACT: PROCEDURE;
/*****************************************************************************/
/* Subroutine VUEACT
/*      View the account data.
/*      If operator knows the secret word, let operator change
/*      the account data for this session.
/*****************************************************************************/

CALL FDV$CDISP( 'ACCOUNT_DATA' );        CALL SRVCHK;
CALL FDV$PUTAL( ACCOUNT_STR );
CALL FDV$PUTD( 'SECRET');
/*
/* This is not the best way to do protection, just a way of showing
/* another FMS feature. At this point, supervisor mode is on, so the
/* only input allowed is to the password field.
/* If operator doesn't know password, return to menu.
/*                                                                    */
CALL FDV$GETAL( , TERMINATOR );          /*Don't care about value now*/
IF TERMINATOR = FDV$K_KP_PER THEN RETURN;
CALL FDV$RET( PASSWORD, 'SECRET' );
IF OPW = PASSWORD THEN RETURN;
/*
/* Allow input from other fields and read from them.
/* If read is terminated by keypad period, don't change account.
/*                                                                    */
CALL FDV$SPOFF;
CALL SIMULATE;  CALL FDV$NDISP; /* Read all fields */
CALL FDV$SPON;                  /* Not really needed, just showing off.*/
IF TERMINATOR = FDV$K_KP_PER
THEN DO;
        CALL FDV$RETAL( ACCOUNT_STR );
        CALL FMTCHK;    /* Update the check workspace */
     END;
END VUEACT;
```

```
SIMULATE: PROCEDURE;
/*****************************************************************************/
/* Simulate action of FDV$GETAL, using FDV$GETAF and PFT.  Could
/* replace this whole routine with a call on FDV$GETAL, but this shows
/* how mainline program can allow same operator freedom of filling in
/* fields but still regain control after each or changed field.
/* Technique is to read any field, looking only at terminator, then do
/* a process field terminator call to do the operator's action.
/* This technique can be used with calls on FDV$GET or FDV$GETAF.
/* This example starts with a GET on field '*', first field on form.
/*****************************************************************************/
DCL     FIELDNAME       CHAR(31),       /*Name of field from FDV$GETAF*/
        FIELDINDEX      FIXED BIN(31);  /*Index of field or named data*/

CALL FDV$GET( JUNK, TERMINATOR, '*' );
CALL FDV$RETFN( FIELDNAME, FIELDINDEX );        /*Get first field's name*/
DO WHILE ('1'B);
    /*
    /* Do any special processing for field FIELDNAME at this point.
    /* ...
    /* Go to next or previous field or leave form
    /*                                                              */
    CALL FDV$PFT( TERMINATOR );
    /*
    /* If status is error, then PFT failed because terminator was
    /* a keypad key, which means return to caller.
    /*                                                              */
    IF FMSSTATUS < 0 THEN RETURN;
    IF TERMINATOR = FDV$K_FT_NTR
    THEN IF FMSSTATUS = 2
         THEN RETURN;
         ELSE DO;
                CALL FDV$PUTL( 'INPUT REQUIRED' );
                CALL FDV$BELL;
              END;
    /*
    /* Go set any other field, returning its name
    /*                                                              */
    CALL FDV$GETAF( JUNK, TERMINATOR, FIELDNAME, FIELDINDEX );
END;
END SIMULATE;
```

```
GETSTA: PROCEDURE;
/*****************************************************************************/
/* Subroutine GETSTA
/* Check FMS status by calling FDV$STAT.
/* If not success (>0), print and stop
/*****************************************************************************/

CALL FDV$STAT( FMSSTATUS, RMSSTATUS );
IF FMSSTATUS > 0 THEN RETURN;
CALL ERROR;      /* and never come back */
END GETSTA;



SRVCHK: PROCEDURE;
/*****************************************************************************/
/* Subroutine SRVCHK
/* Check FMS status by looking at the status recording variables.
/*****************************************************************************/

IF FMSSTATUS > 0 THEN RETURN;
CALL ERROR;      /* and never come back */
END SRVCHK;



ERROR: PROCEDURE;
/*****************************************************************************/
/* There is an error returned in the status variables. Detach the
/* terminal to clean up, then print the errors, and stop.
/*****************************************************************************/

CALL FDV$DTERM( TCA );
PUT SKIP LIST( 'FDV ERROR.');
PUT SKIP LIST( '','FMS STATUS:',FMSSTATUS);
PUT SKIP LIST( '','RMS STATUS:',RMSSTATUS);
STOP;
END ERROR;
```

```
CENTS: PROCEDURE ( PENNIES ) RETURNS( CHAR(*) );
/*****************************************************************************/
/* CENTS
/*      Takes a parameter representing cents and converts to a numeric string
/*      suitable to outputting into a field six wide with two decimal digits.
/*      The important thing to note is that numbers less than 100 should be
/*      converted with leading zeros so we don't end up outputting a string
/*      like "bbbbb0" which then ends up on the screen like "bbbb.b0".
/*****************************************************************************/

DCL PENNIES             FIXED BIN( 31 );
DCL CHAR_PENNIES        PICTURE 'ZZZ999';

CHAR_PENNIES = PENNIES;
RETURN ( CHAR_PENNIES );

END CENTS;

END SAMP;



/*****************************************************************************/
/* The following routines are external to SAMP.  They are UARs, called by the
/* Form Driver to process field completions or function keys.
/*****************************************************************************/

VALID1: PROCEDURE RETURNS(FIXED BIN(31));
/*****************************************************************************/
/* VALID1
/*      UAR for field validation of any one character field. The
/*      UAR associated data has in it the legal characters allowed,
/*      except that blank is not allowed unless it appears before
/*      the first trailing blank.  For example an assoc. value string
/*      'aqr' implies that only the letters a, q, and r are allowed.
/*      A string ' aqr' means that blank is acceptable in addition
/*      to a, q, and r. Note that this routine is case sensitive
/*      (that is, it checks for correct case).  You can set around
/*      case sensitivity by using the force upper case field attribute
/*      and putting only capitals into the UAR associated value
/*      string.
/*
```

```
/*      This routine can be used with any form and field since
/*      it determines the context for itself.
/*****************************************************************************/

%INCLUDE 'FDVDEFCAL.PLI';

DCL UARVAL      CHAR(80),
    FRMNAM      CHAR(31),
    FLDNAME     CHAR(31),
    FVALUE      CHAR(1),
    (CURPOS,FLDTRM,INSOVR,FINDEX,HELPNUM) FIXED BIN(31),
    (WKSP,TCA)  POINTER;

DCL TRIM EXTERNAL ENTRY( CHAR(*) ) RETURNS(CHAR(*));

/*
/* Retrieve context: we will ignore TCA address, WKSP address, FRMNAM,
/*      CURPOS, FLDTRM, INSOVR and HELPNUM, using only UARVAL, and only the
/*      initial, non-blank characters of it.
/* Retrieve field name and index.
/* Retrieve field value.                                                  */
CALL FDV$RETCX( TCA, WKSP, FRMNAM, UARVAL, CURPOS, FLDTRM, INSOVR, HELPNUM );
CALL FDV$RETFN( FLDNAME, FINDEX );
CALL FDV$RET( FVALUE, FLDNAME, FINDEX );

/*
/* To be valid, FVALUE must occur in the string UARVAL
/*                                                                        */
IF INDEX( UARVAL, FVALUE ) > 0
THEN RETURN( FDV$K_UVAL_SUC );                  /*Success*/
ELSE DO;
        CALL FDV$PUTL( 'Illegal value' );
        RETURN( FDV$K_UVAL_FAIL );
     END;
END VALID1;
```

```
TAKE15: PROCEDURE RETURNS(FIXED BIN(31));
/*****************************************************************************/
/* Subroutine TAKE15:
/* Function Key User Action Routine for the MENU form of SAMP.
/* Convert Keypad 1-5 into field values 1-5.
/* Convert Keypad period into field value 1.
/* Reject all other function keys with error message.
/*****************************************************************************/
%INCLUDE 'FDVDEFCAL.PLI';

DCL (VALUE,UARVAL)      CHAR(1),
    FRMNAM              CHAR(4),
    (CURPOS,INSOVR,FLDTRM,HELPNUM) FIXED BIN(31),
    (WKSP,TCA)          POINTER;
/*
/* Retrieve context: we will ignore TCA address, WKSP address, FRMNAM,
/* UARVAL, CURPOS, INSOVR and HELPNUM, using only FLDTRM
/*                                                                    */
CALL FDV$RETCX( TCA, WKSP, FRMNAM, UARVAL, CURPOS, FLDTRM, INSOVR, HELPNUM );
/*
/* Do the conversion, displaying the value converted if found.
/* Reject if not one of the expected terminators.                     */
VALUE = '';
IF FLDTRM = FDV$K_KP_1 THEN VALUE = '1';
IF FLDTRM = FDV$K_KP_2 THEN VALUE = '2';
IF FLDTRM = FDV$K_KP_3 THEN VALUE = '3';
IF FLDTRM = FDV$K_KP_4 THEN VALUE = '4';
IF FLDTRM = FDV$K_KP_5 THEN VALUE = '5';
IF FLDTRM = FDV$K_KP_PER THEN VALUE = '1';
IF VALUE = ''
THEN DO;
        CALL FDV$PUT( VALUE, 'OPTION' );
        /* Treat as if it is RETURN */
        RETURN( FDV$K_UKEY_NTR );
     END;
ELSE DO;
        CALL FDV$PUTL( 'Illegal Function Key' );
        CALL FDV$SIGOP;
        /* Just ignore it now */
        RETURN( FDV$K_UKEY_SUC );
     END;
END TAKE15;
```

```
PASSKY: PROCEDURE RETURNS(FIXED BIN(31));
/*****************************************************************************/
/* General function key var to pass only those from the (small) list         */
/* in the var associated value string and reject all others.                 */
/* The list is of the form: n <oneblank> n <oneblank> ... n <manyblanks>      */
/* For example the string '110 112' would accept keypad period and           */
/* keypad zero but no other function keys.                                   */
/*****************************************************************************/

%INCLUDE 'FDVDEFCAL.PLI';

DCL UARVAL      CHAR(82),
    FRMNAM      CHAR(31),
    (CURPOS,INSOVR,FLDTRM,NONBLANK,NEXTBLANK,HELPNUM) FIXED BIN(31),
    (WKSP,TCA)  POINTER;
/*
/* Retrieve context: we will ignore TCA address, WKSP address, FRMNAM,
/*       INSOVR, HELPNUM and CURPOS, using only FLDTRM and UARVAL. */
CALL FDV$RETCX( TCA, WKSP, FRMNAM, UARVAL, CURPOS, FLDTRM, INSOVR, HELPNUM );

/*
/* Break up the list into numbers.  Check each against the actual
/* terminator. If terminator found in list, return success.
/*                                                                  */
NONBLANK = 1;            /* Beginning of string */
DO WHILE (SUBSTR( UARVAL, NONBLANK, 1) =  ' ' );
    NEXTBLANK = INDEX( SUBSTR(UARVAL, NONBLANK ), ' ') + NONBLANK - 1;
    IF FLDTRM = FIXED( SUBSTR( UARVAL, NONBLANK, NEXTBLANK - 1 ), 31 )
    THEN RETURN( FDV$K_UKEY_TRM );      /*Pass key to application*/
    NONBLANK = NEXTBLANK + 1;
END;
RETURN( FDV$K_UKEY_ERR );               /*Let FDV do the beeping*/
END PASSKY;
```

```
CHKCHK: PROCEDURE RETURNS(FIXED BIN(31));
/*****************************************************************************/
/* UAR for SAMP CHECK form.  Makes sure that the check amount is
/* less than or equal to the current balance.  If not, complain and
/* change video attributes on balance field so the potential bouncer
/* can see what there is to work with.
/*****************************************************************************/

%INCLUDE 'FDVDEFCAL.PLI';

DCL (BALANCE,AMTPAY)    CHAR(6),
    BLINKBOLD           FIXED BIN(31);

CALL FDV$RET( BALANCE, 'BALANCE' );
CALL FDV$RET( AMTPAY, 'AMTPAY' );
IF FIXED( BALANCE, 31 ) >= FIXED( AMTPAY, 31 )
THEN DO;
        BLINKBOLD = -1;                             /*Restore to original*/
        CALL FDV$AFVA( BLINKBOLD, 'BALANCE' );
        RETURN( FDV$K_UVAL_SUC );
     END;
ELSE DO;
        BLINKBOLD = 3;                              /*Make it very visible  */
        CALL FDV$AFVA( BLINKBOLD, 'BALANCE' );
        CALL FDV$PUTL( 'Your balance doesn''t cover that much, reenter amount' );
        RETURN( FDV$K_UVAL_FAIL );
     END;
END CHKCHK;



RANGE: PROCEDURE RETURNS(FIXED BIN(31));
/*****************************************************************************/
/* General purpose UAR to check the range of any numeric item.  The
/* associated UAR data must have one of the four forms:
/*      L,U<space>{message}
/*      ,U<space>{message}
/*      L,<space>{message}
/*      ,<space>{message}
/* where L is lower bound, U is upper bound, and {message} is an
/* optional error message in case the field value is out of bounds.
```

```
/* If one of the bounds isn't given, it isn't checked for.  If neither
/* bound is given, nothing is checked, everything succeeds.  If the
/* UAR value doesn't have a comma, a FDV$_UAR error message is returned
/* to the calling program by the FDV so the form designer has to go
/* back and do it right. If no (message) is given, a simple
/* "out of range U:L" message is given to the hapless operator.
/*
/* This UAR can work with any form and numeric field since it gets
/* context itself.  Care must be taken with fields using field marker
/* periods since those periods are not returned to the program.
/*****************************************************************************/

%INCLUDE 'FDVDEFCAL.PLI';

DCL UARVAL                  CHAR(80),
    (FRMNAM, NAME)          CHAR(31),
    CNUMBER                 CHAR(132),
    (CURPOS,INSOVR,NUMBER,FLDTRM,HELPNUM,
    INTEGER_INDEX,COMMA,BLANK)          FIXED BIN(31),
    (WKSP,TCA)              POINTER;
DCL TRIM                    EXTERNAL ENTRY( CHAR(*) ) RETURNS(CHAR(*));
/*
/* Get context which yields associated data value (ignore other stuff).
/* Get current field name and index.
/* Get field value.
/*                                                                    */
CALL FDV$RETCX( TCA, WKSP, FRMNAM, UARVAL, CURPOS, FLDTRM, INSOVR, HELPNUM );
CALL FDV$RETFN( NAME, INTEGER_INDEX );
CALL FDV$RET( CNUMBER, NAME, INTEGER_INDEX );
NUMBER = FIXED( CNUMBER, 31 );
/*
/* Find comma and blank delimiters.
/* Check for lower bound.
/*                                                                    */
COMMA = INDEX( UARVAL, ',');
BLANK = INDEX( SUBSTR(UARVAL,COMMA+1),' ') + COMMA;
IF COMMA = 0
THEN RETURN(0);                     /* Illegal UARVAL string, FDV returns error */
IF COMMA = 1
THEN IF NUMBER < FIXED( SUBSTR( UARVAL, 1, COMMA - 1 ), 31 )
     THEN GOTO ERROR;
```

```
/*
/* Check for upper bound
/*                                  */
IF BLANK = COMMA + 1
THEN IF NUMBER > FIXED( SUBSTR( UARVAL, COMMA + 1, BLANK - 1 - COMMA), 31 )
     THEN GOTO ERROR;
/*
/* Passed both tests successfully, return success for UAR value
/*                                          */
RETURN( FDV$K_UVAL_SUC );
/*
/* Give error message: either from the UARVAL or make one up.
/*                                                              */
ERROR:
IF SUBSTR( UARVAL, BLANK + 1, 1 ) = ' '
THEN CALL FDV$PUTL( SUBSTR( UARVAL, BLANK + 1, 80-BLANK ) );
ELSE CALL FDV$PUTL( 'Field value out of bounds. Must be in range "'
                    || SUBSTR( UARVAL, 1, BLANK - 1 ) || '".' );
CALL FDV$SIGOP;                                 /*Beep, too. */
RETURN(FDV$K_UVAL_FAIL);
END RANGE;



TRIM: PROCEDURE( INPUT_STRING ) RETURNS( CHAR(*) );
/*****************************************************************************/
/* Function TRIM
/* This function returns the parameter character string with trailing blanks
/* removed.
/*****************************************************************************/

DCL  INPUT_STRING         CHAR(*),
     I                    FIXED BIN(15);

I = LENGTH( INPUT_STRING );
DO WHILE ( I = 0  SUBSTR( INPUT_STRING, I, 1 ) = ' ' );
     I = I - 1;
END;

RETURN( SUBSTR( INPUT_STRING, 1, I ));
END TRIM;
```

```
/*****************************************************************************/
/* FDVDEFCAL.PLI  --  This is the include file for FMS applications */
/*                    using PL/I with all FDV references as calls.  */
/*****************************************************************************/

/*************************/
/* FMS terminator codes: */
/*************************/
%REPLACE FDV$K_FT_NTR     BY  0; /*Enter (i.e. end GETs)*/
%REPLACE FDV$K_FT_NXT     BY  1; /*Next field */
%REPLACE FDV$K_FT_PRV     BY  2; /*Previous field */
%REPLACE FDV$K_FT_ATB     BY  3; /*Automatically move to next field*/
%REPLACE FDV$K_FT_XBK     BY  4; /*Exit scrolled area backward*/
%REPLACE FDV$K_FT_XFW     BY  5; /*Exit scrolled area forward*/
%REPLACE FDV$K_FT_SNX     BY  6; /*Scroll forward to next field */
%REPLACE FDV$K_FT_SPR     BY  7; /*Scroll backward to previous field*/
%REPLACE FDV$K_FT_SFW     BY  8; /*Scroll forward*/
%REPLACE FDV$K_FT_SBK     BY  9; /*Scroll backward*/
%REPLACE FDV$K_FT_ILG_NXT BY 11; /*Illegal context for next field */
%REPLACE FDV$K_FT_ILG_PRV BY 12; /*Illegal context for previous field */
%REPLACE FDV$K_FT_ILG_ATB BY 13; /*Illegal context for auto move to next field*/
%REPLACE FDV$K_FT_ILG_XBK BY 14; /*Illegal context for exit scrolled area backward*/
%REPLACE FDV$K_FT_ILG_XFW BY 15; /*Illegal context for exit scrolled area forward*/
%REPLACE FDV$K_FT_ILG_SFW BY 16; /*Illegal context for scroll forward*/
%REPLACE FDV$K_FT_ILG_SBK BY 17; /*Illegal context for scroll backward*/
/*************************************************************/
/* Function key terminators returned from GETs and WAIT */
/* Also used as FDV keycodes for use with DFKBD.  */
/*************************************************************/
%REPLACE FDV$K_AR_UP      BY   99;
%REPLACE FDV$K_AR_DOWN    BY  100;
%REPLACE FDV$K_AR_LEFT    BY  101;
%REPLACE FDV$K_AR_RIGHT   BY  102;
%REPLACE FDV$K_PF_1       BY  103;
%REPLACE FDV$K_PF_2       BY  104;
%REPLACE FDV$K_PF_3       BY  105;
%REPLACE FDV$K_PF_4       BY  106;
%REPLACE FDV$K_KP_NTR     BY  107;
%REPLACE FDV$K_KP_COM     BY  108;
%REPLACE FDV$K_KP_HYP     BY  109;
%REPLACE FDV$K_KP_PER     BY  110;
%REPLACE FDV$K_KP_0       BY  112;
```

```
%REPLACE FDV$K_KP_1       BY  113;
%REPLACE FDV$K_KP_2       BY  114;
%REPLACE FDV$K_KP_3       BY  115;
%REPLACE FDV$K_KP_4       BY  116;
%REPLACE FDV$K_KP_5       BY  117;
%REPLACE FDV$K_KP_6       BY  118;
%REPLACE FDV$K_KP_7       BY  119;
%REPLACE FDV$K_KP_8       BY  120;
%REPLACE FDV$K_KP_9       BY  121;
%REPLACE FDV$K_GAR_UP     BY  227;
%REPLACE FDV$K_GAR_DOWN   BY  228;
%REPLACE FDV$K_GAR_RIGHT  BY  229;
%REPLACE FDV$K_GAR_LEFT   BY  230;
%REPLACE FDV$K_GPF_1      BY  231;
%REPLACE FDV$K_GPF_2      BY  232;
%REPLACE FDV$K_GPF_3      BY  233;
%REPLACE FDV$K_GPF_4      BY  234;
%REPLACE FDV$K_GKP_NTR    BY  235;
%REPLACE FDV$K_GKP_COM    BY  236;
%REPLACE FDV$K_GKP_HYP    BY  237;
%REPLACE FDV$K_GKP_PER    BY  238;
%REPLACE FDV$K_GKP_0      BY  240;
%REPLACE FDV$K_GKP_1      BY  241;
%REPLACE FDV$K_GKP_2      BY  242;
%REPLACE FDV$K_GKP_3      BY  243;
%REPLACE FDV$K_GKP_4      BY  244;
%REPLACE FDV$K_GKP_5      BY  245;
%REPLACE FDV$K_GKP_6      BY  246;
%REPLACE FDV$K_GKP_7      BY  247;
%REPLACE FDV$K_GKP_8      BY  248;
%REPLACE FDV$K_GKP_9      BY  249;
/*********************************************/
/* FDV Keyfunctions. For use in DFKBD call. */
/*********************************************/
%REPLACE FDV$K_KF_GOLD   BY   1;
%REPLACE FDV$K_KF_RESET  BY   2;
%REPLACE FDV$K_KF_CRSLF  BY   3;
%REPLACE FDV$K_KF_CRSRT  BY   4;
%REPLACE FDV$K_KF_DLCHR  BY   5;
%REPLACE FDV$K_KF_DLFLD  BY   6;
%REPLACE FDV$K_KF_INS    BY   7;
%REPLACE FDV$K_KF_OVR    BY   8;
```

```
%REPLACE FDV$K_KF_RFRSH BY   9;
%REPLACE FDV$K_KF_HELP  BY  10;
%REPLACE FDV$K_KF_NXT   BY  11;
%REPLACE FDV$K_KF_PRV   BY  12;
%REPLACE FDV$K_KF_NTR   BY  13;
%REPLACE FDV$K_KF_SBK   BY  14;
%REPLACE FDV$K_KF_SFW   BY  15;
%REPLACE FDV$K_KF_XBK   BY  16;
%REPLACE FDV$K_KF_XFW   BY  17;
%REPLACE FDV$K_KF_NONE  BY   0;
%REPLACE FDV$K_KF_DFLT  BY  -1;
/*************************************************************/
/* UAR return codes. These codes are returned by UAR to FDV. */
/*************************************************************/
/* Field completion return codes */
/*********************************/
%REPLACE FDV$K_UVAL_SUC     BY  1000;   /*Field completion success */
%REPLACE FDV$K_UVAL_FAIL    BY  1001;   /*Field completion failure      */
%REPLACE FDV$K_UVAL_END     BY  1002;   /*Field completion suc-stop UARs*/
/*************************/
/* Help UAR return codes */
/*************************/
%REPLACE FDV$K_UHELP_NO     BY  2000;   /*No help given, try next step  */
%REPLACE FDV$K_UHELPED      BY  2001;   /*Help given, continue sequence */
%REPLACE FDV$K_UHELP_ALL    BY  2002;   /*Help given, repeat UAR        */
/***********************************/
/* Function Key UAR return codes */
/***********************************/
%REPLACE FDV$K_UKEY_ERR     BY  3000;   /*Fn Key failure, FDV signals   */
%REPLACE FDV$K_UKEY_TRM     BY  3001;   /*Fn Key success, normal F.K.   */
%REPLACE FDV$K_UKEY_NXT     BY  3002;   /*Fn Key succ, treat as NEXT    */
%REPLACE FDV$K_UKEY_NTR     BY  3003;   /*Fn Key succ, treat as ENTER   */
%REPLACE FDV$K_UKEY_SUC     BY  3004;   /*Fn Key succ, ignore           */
/*************************************************************************/
/* FDV status codes returned when FDV$... routines are called as functions. */
/* These codes are VMS status codes and can be signalled.  They correspond  */
/* one-to-one with the FMS status codes retrievable from FDV$STAT.          */
/*************************************************************************/
%REPLACE FDV$_SUC BY 2719889 ;
%REPLACE FDV$_INC BY 2719897 ;
%REPLACE FDV$_MOD BY 2719905 ;
%REPLACE FDV$_IMP BY 2719922 ;
```

```
%REPLACE FDV$_FSP BY 2719930 ;
%REPLACE FDV$_IOL BY 2719938 ;
%REPLACE FDV$_FLB BY 2719946 ;
%REPLACE FDV$_ICH BY 2719954 ;
%REPLACE FDV$_FCH BY 2719962 ;
%REPLACE FDV$_FRM BY 2719970 ;
%REPLACE FDV$_FNM BY 2719978 ;
%REPLACE FDV$_LIN BY 2719986 ;
%REPLACE FDV$_FLD BY 2719994 ;
%REPLACE FDV$_NCF BY 2720002 ;
%REPLACE FDV$_DSP BY 2720010 ;
%REPLACE FDV$_NSC BY 2720018 ;
%REPLACE FDV$_DNM BY 2720026 ;
%REPLACE FDV$_DLN BY 2720034 ;
%REPLACE FDV$_UTR BY 2720042 ;
%REPLACE FDV$_IOR BY 2720050 ;
%REPLACE FDV$_IFN BY 2720058 ;
%REPLACE FDV$_ARG BY 2720066 ;
%REPLACE FDV$_INI BY 2720074 ;
%REPLACE FDV$_STR BY 2720082 ;
%REPLACE FDV$_IVM BY 2720090 ;
%REPLACE FDV$_FVM BY 2720098 ;
%REPLACE FDV$_ITT BY 2720106 ;
%REPLACE FDV$_TCA BY 2720114 ;
%REPLACE FDV$_STA BY 2720122 ;
%REPLACE FDV$_WID BY 2720130 ;
%REPLACE FDV$_NFL BY 2720138 ;
%REPLACE FDV$_IBF BY 2720146 ;
%REPLACE FDV$_NDS BY 2720154 ;
%REPLACE FDV$_UDP BY 2720162 ;
%REPLACE FDV$_UAR BY 2720170 ;
%REPLACE FDV$_UNF BY 2720178 ;
%REPLACE FDV$_CAN BY 2720194 ;
%REPLACE FDV$_KIF BY 2720202 ;
%REPLACE FDV$_KEX BY 2720210 ;
%REPLACE FDV$_KTW BY 2720218 ;
%REPLACE FDV$_KIL BY 2720226 ;
%REPLACE FDV$_TMO BY 2720234 ;
%REPLACE FDV$_LLI BY 2720242 ;
%REPLACE FDV$_VAL BY 2720250 ;
%REPLACE FDV$_IFU BY 2720258 ;
%REPLACE FDV$_SYS BY 2720266 ;
```

```
/*****************************************************************************/
/* FMS status codes returned when FDV$STAT routine is called.                */
/*****************************************************************************/
/* Success codes. */

%REPLACE FDV$K_SUC BY   1;
%REPLACE FDV$K_INC BY   2;
%REPLACE FDV$K_MOD BY   3;

/* Failure code */

%REPLACE FDV$K_IMP BY   -2;
%REPLACE FDV$K_FSP BY   -3;
%REPLACE FDV$K_IDL BY   -4;
%REPLACE FDV$K_FLB BY   -5;
%REPLACE FDV$K_ICH BY   -6;
%REPLACE FDV$K_FCH BY   -7;
%REPLACE FDV$K_FRM BY   -8;
%REPLACE FDV$K_FNM BY   -9;
%REPLACE FDV$K__IN BY   -10;
%REPLACE FDV$K_FLD BY   -11;
%REPLACE FDV$K_NOF BY   -12;
%REPLACE FDV$K_DSP BY   -13;
%REPLACE FDV$K_NSC BY   -14;
%REPLACE FDV$K_DNM BY   -15;
%REPLACE FDV$K_DLN BY   -16;
%REPLACE FDV$K_UTR BY   -17;
%REPLACE FDV$K_ICR BY   -18;
%REPLACE FDV$K_IFN BY   -19;
%REPLACE FDV$K_ARG BY   -20;
%REPLACE FDV$K_INI BY   -21;
%REPLACE FDV$K_STR BY   -22;
%REPLACE FDV$K_FVM BY   -23;
%REPLACE FDV$K_IVM BY   -24;
%REPLACE FDV$K_ITT BY   -25;
%REPLACE FDV$K_TCA BY   -26;
%REPLACE FDV$K_STA BY   -27;
%REPLACE FDV$K_WID BY   -28;
%REPLACE FDV$K_NFL BY   -29;
%REPLACE FDV$K_IBF BY   -30;
%REPLACE FDV$K_NDS BY   -31;
%REPLACE FDV$K_UDP BY   -33;
```

```
%REPLACE FDV$K_UAR BY   -34;
%REPLACE FDV$K_UNF BY   -35;
%REPLACE FDV$K_CAN BY   -39;
%REPLACE FDV$K_KIF BY   -40;
%REPLACE FDV$K_KEX BY   -41;
%REPLACE FDV$K_KTW BY   -42;
%REPLACE FDV$K_KIL BY   -43;
%REPLACE FDV$K_TMO BY   -44;
%REPLACE FDV$K_LLI BY   -45;
%REPLACE FDV$K_VAL BY   -47;
%REPLACE FDV$K_IFU BY   -48;
%REPLACE FDV$K_SYS BY   -49;

/************************************************************/
/* Declare the FDV routines' calling sequence.              */
/* This is the sequence for CALLs.  For calling FDV as      */
/* functions, add "RETURNS(FIXED BIN)" to each line.        */
/************************************************************/
DCL FDV$ADLVA    EXTERNAL ENTRY( FIXED BIN );
DCL FDV$AFVA     EXTERNAL ENTRY( FIXED BIN, CHAR(*), FIXED BIN )                  OPTIONS(VARIABLE);
DCL FDV$ATERM    EXTERNAL ENTRY( (*)FIXED BIN, FIXED BIN, FIXED BIN, CHAR(*) )    OPTIONS(VARIABLE);
DCL FDV$AWKSP    EXTERNAL ENTRY( (*)FIXED BIN, FIXED BIN );
DCL FDV$BELL     EXTERNAL ENTRY();
DCL FDV$CANCL    EXTERNAL ENTRY();
DCL FDV$CDISP    EXTERNAL ENTRY( CHAR(*), FIXED BIN )                             OPTIONS(VARIABLE);
DCL FDV$CLEAR    EXTERNAL ENTRY( FIXED BIN, FIXED BIN )                           OPTIONS(VARIABLE);
DCL FDV$DEL      EXTERNAL ENTRY( CHAR(*) );
DCL FDV$DFKBD    EXTERNAL ENTRY( (*)FIXED BIN(15), FIXED BIN );
DCL FDV$DISP     EXTERNAL ENTRY( CHAR(*), FIXED BIN )                             OPTIONS(VARIABLE);
DCL FDV$DISPW    EXTERNAL ENTRY( FIXED BIN )                                      OPTIONS(VARIABLE);;
DCL FDV$DPCOM    EXTERNAL ENTRY( FIXED BIN );
DCL FDV$DTERM    EXTERNAL ENTRY( (*)FIXED BIN );
DCL FDV$DWKSP    EXTERNAL ENTRY( (*)FIXED BIN );
DCL FDV$GET      EXTERNAL ENTRY( CHAR(*), FIXED BIN, CHAR(*), FIXED BIN )         OPTIONS(VARIABLE);
DCL FDV$GETAF    EXTERNAL ENTRY( CHAR(*), FIXED BIN, CHAR(*), FIXED BIN )         OPTIONS(VARIABLE);
DCL FDV$GETAL    EXTERNAL ENTRY( CHAR(*), FIXED BIN, CHAR(*), FIXED BIN )         OPTIONS(VARIABLE);
DCL FDV$GETDL    EXTERNAL ENTRY( CHAR(*), FIXED BIN, FIXED BIN, CHAR(*) )         OPTIONS(VARIABLE);
DCL FDV$GETSC    EXTERNAL ENTRY( CHAR(*), CHAR(*), FIXED BIN )                    OPTIONS(VARIABLE);
DCL FDV$ILTRM    EXTERNAL ENTRY( FIXED BIN );
DCL FDV$LCHAN    EXTERNAL ENTRY( FIXED BIN );
DCL FDV$LCLOS    EXTERNAL ENTRY();
DCL FDV$LEDOF    EXTERNAL ENTRY( FIXED BIN );
```

```
DCL FDV$LEDON    EXTERNAL ENTRY( FIXED BIN );
DCL FDV$LOAD     EXTERNAL ENTRY( CHAR(*) );
DCL FDV$LOPEN    EXTERNAL ENTRY( CHAR(*), FIXED BIN )                                    OPTIONS(VARIABLE);
DCL FDV$NDISP    EXTERNAL ENTRY();
DCL FDV$PFT      EXTERNAL ENTRY( FIXED BIN, CHAR(*), CHAR(*), CHAR(*), FIXED BIN ) OPTIONS(VARIABLE);
DCL FDV$PUT      EXTERNAL ENTRY( CHAR(*), CHAR(*), FIXED BIN )                           OPTIONS(VARIABLE);
DCL FDV$PUTAL    EXTERNAL ENTRY( CHAR(*) );
DCL FDV$PUTD     EXTERNAL ENTRY( CHAR(*), FIXED BIN )                                    OPTIONS(VARIABLE);
DCL FDV$PUTDA    EXTERNAL ENTRY();
DCL FDV$PUTL     EXTERNAL ENTRY( CHAR(*), FIXED BIN )                                    OPTIONS(VARIABLE);
DCL FDV$PUTSC    EXTERNAL ENTRY( CHAR(*), CHAR(*) )                                      OPTIONS(VARIABLE);
DCL FDV$READ     EXTERNAL ENTRY( CHAR(*), (*)FIXED BIN, FIXED BIN, FIXED BIN );
DCL FDV$RET      EXTERNAL ENTRY( CHAR(*), CHAR(*), FIXED BIN )                           OPTIONS(VARIABLE);
DCL FDV$RETAL    EXTERNAL ENTRY( CHAR(*) );
DCL FDV$RETCX    EXTERNAL ENTRY( POINTER, POINTER, CHAR(*), CHAR(*) , FIXED BIN,
                                 FIXED BIN, FIXED BIN, FIXED BIN );
DCL FDV$RETDI    EXTERNAL ENTRY( FIXED BIN, CHAR(*), CHAR(*) )                           OPTIONS(VARIABLE);
DCL FDV$RETDN    EXTERNAL ENTRY( CHAR(*) , CHAR(*), FIXED BIN )                          OPTIONS(VARIABLE);
DCL FDV$RETFL    EXTERNAL ENTRY( FIXED BIN, CHAR(*), FIXED BIN, FIXED BIN )              OPTIONS(VARIABLE);
DCL FDV$RETFN    EXTERNAL ENTRY( CHAR(*), FIXED BIN )                                    OPTIONS(VARIABLE);
DCL FDV$RETFO    EXTERNAL ENTRY( FIXED BIN, CHAR(*), FIXED BIN );
DCL FDV$RETLE    EXTERNAL ENTRY( FIXED BIN, CHAR(*), FIXED BIN )                         OPTIONS(VARIABLE);
DCL FDV$RFRSH    EXTERNAL ENTRY();
DCL FDV$SIGOF    EXTERNAL ENTRY();
DCL FDV$SPADA    EXTERNAL ENTRY( FIXED BIN );
DCL FDV$SPOFF    EXTERNAL ENTRY();
DCL FDV$SPON     EXTERNAL ENTRY();
DCL FDV$SSIGQ    EXTERNAL ENTRY( FIXED BIN );
DCL FDV$SSRV     EXTERNAL ENTRY( FIXED BIN, FIXED BIN )                                  OPTIONS(VARIABLE);
DCL FDV$STAT     EXTERNAL ENTRY( FIXED BIN, FIXED BIN )                                  OPTIONS(VARIABLE);
DCL FDV$SWKSP    EXTERNAL ENTRY( (*)FIXED BIN )                                          OPTIONS(VARIABLE);
DCL FDV$WAIT     EXTERNAL ENTRY( FIXED BIN )                                             OPTIONS(VARIABLE);
/*****************************************************************************/
/* FDVDEFFNC.PLI  --  This is the include file for FMS applications    */
/*                    using PL/I with all FDV references as functions. */
/*****************************************************************************/
/*************************/
/* FMS terminator codes: */
/*************************/
%REPLACE FDV$K_FT_NTR     BY  0;        /*Enter (i.e. end GETs)*/
%REPLACE FDV$K_FT_NXT     BY  1;        /*Next field */
%REPLACE FDV$K_FT_PRV     BY  2;        /*Previous field */
```

```
%REPLACE FDV$K_FT_ATB      BY  3;        /*Automatically move to next field*/
%REPLACE FDV$K_FT_XBK      BY  4;        /*Exit scrolled area backward*/
%REPLACE FDV$K_FT_XFW      BY  5;        /*Exit scrolled area forward*/
%REPLACE FDV$K_FT_SNX      BY  6;        /*Scroll forward to next field */
%REPLACE FDV$K_FT_SPR      BY  7;        /*Scroll backward to previous field*/
%REPLACE FDV$K_FT_SFW      BY  8;        /*Scroll forward*/
%REPLACE FDV$K_FT_SBK      BY  9;        /*Scroll backward*/
%REPLACE FDV$K_FT_ILG_NXT  BY 11;        /*Illegal context for next field */
%REPLACE FDV$K_FT_ILG_PRV  BY 12;        /*Illegal context for previous field */
%REPLACE FDV$K_FT_ILG_ATB  BY 13;        /*Illegal context for auto move to next field*/
%REPLACE FDV$K_FT_ILG_XBK  BY 14;        /*Illegal context for exit scrolled area backward*/
%REPLACE FDV$K_FT_ILG_XFW  BY 15;        /*Illegal context for exit scrolled area forward*/
%REPLACE FDV$K_FT_ILG_SFW  BY 16;        /*Illegal context for scroll forward*/
%REPLACE FDV$K_FT_ILG_SBK  BY 17;        /*Illegal context for scroll backward*/
/*************************************************************/
/* Function Key terminators returned from GETs and WAIT */
/* Also used as FDV keycodes for use with DFKBD.         */
/*************************************************************/
%REPLACE FDV$K_AR_UP       BY  99;
%REPLACE FDV$K_AR_DOWN     BY 100;
%REPLACE FDV$K_AR_LEFT     BY 101;
%REPLACE FDV$K_AR_RIGHT    BY 102;
%REPLACE FDV$K_PF_1        BY 103;
%REPLACE FDV$K_PF_2        BY 104;
%REPLACE FDV$K_PF_3        BY 105;
%REPLACE FDV$K_PF_4        BY 106;
%REPLACE FDV$K_KP_ENTR     BY 107;
%REPLACE FDV$K_KP_COM      BY 108;
%REPLACE FDV$K_KP_HYP      BY 109;
%REPLACE FDV$K_KP_PER      BY 110;
%REPLACE FDV$K_KP_0        BY 112;
%REPLACE FDV$K_KP_1        BY 113;
%REPLACE FDV$K_KP_2        BY 114;
%REPLACE FDV$K_KP_3        BY 115;
%REPLACE FDV$K_KP_4        BY 116;
%REPLACE FDV$K_KP_5        BY 117;
%REPLACE FDV$K_KP_6        BY 118;
%REPLACE FDV$K_KP_7        BY 119;
%REPLACE FDV$K_KP_8        BY 120;
%REPLACE FDV$K_KP_9        BY 121;
%REPLACE FDV$K_GAR_UP      BY 227;
%REPLACE FDV$K_GAR_DOWN    BY 228;
```

```
%REPLACE FDV$K_GAR_RIGHT BY  229;
%REPLACE FDV$K_GAR_LEFT  BY  230;
%REPLACE FDV$K_GPF_1     BY  231;
%REPLACE FDV$K_GPF_2     BY  232;
%REPLACE FDV$K_GPF_3     BY  233;
%REPLACE FDV$K_GPF_4     BY  234;
%REPLACE FDV$K_GKP_NTR   BY  235;
%REPLACE FDV$K_GKP_COM   BY  236;
%REPLACE FDV$K_GKP_HYP   BY  237;
%REPLACE FDV$K_GKP_PER   BY  238;
%REPLACE FDV$K_GKP_0     BY  240;
%REPLACE FDV$K_GKP_1     BY  241;
%REPLACE FDV$K_GKP_2     BY  242;
%REPLACE FDV$K_GKP_3     BY  243;
%REPLACE FDV$K_GKP_4     BY  244;
%REPLACE FDV$K_GKP_5     BY  245;
%REPLACE FDV$K_GKP_6     BY  246;
%REPLACE FDV$K_GKP_7     BY  247;
%REPLACE FDV$K_GKP_8     BY  248;
%REPLACE FDV$K_GKP_9     BY  249;
/*********************************************/
/* FDV Keyfunctions. For use in DFKBD call. */
/*********************************************/
%REPLACE FDV$K_KF_GOLD   BY   1;
%REPLACE FDV$K_KF_RESET  BY   2;
%REPLACE FDV$K_KF_CRSLF  BY   3;
%REPLACE FDV$K_KF_CRSRT  BY   4;
%REPLACE FDV$K_KF_DLCHR  BY   5;
%REPLACE FDV$K_KF_DLFLD  BY   6;
%REPLACE FDV$K_KF_INS    BY   7;
%REPLACE FDV$K_KF_OVR    BY   8;
%REPLACE FDV$K_KF_RFRSH  BY   9;
%REPLACE FDV$K_KF_HELP   BY  10;
%REPLACE FDV$K_KF_NXT    BY  11;
%REPLACE FDV$K_KF_PRV    BY  12;
%REPLACE FDV$K_KF_NTR    BY  13;
%REPLACE FDV$K_KF_SBK    BY  14;
%REPLACE FDV$K_KF_SFW    BY  15;
%REPLACE FDV$K_KF_XBK    BY  16;
%REPLACE FDV$K_KF_XFW    BY  17;
%REPLACE FDV$K_KF_NONE   BY   0;
%REPLACE FDV$K_KF_DFLT   BY  -1;
```

```
/*****************************************************************/
/* UAR return codes. These codes are returned by UAR to FDV. */
/*****************************************************************/
/* Field completion return codes */
/*********************************/
%REPLACE FDV$K_UVAL_SUC     BY  1000;   /*Field completion success      */
%REPLACE FDV$K_UVAL_FAIL    BY  1001;   /*Field completion failure      */
%REPLACE FDV$K_UVAL_END     BY  1002;   /*Field completion suc-stop UARs*/
/*************************/
/* Help UAR return codes */
/*************************/
%REPLACE FDV$K_UHELP_NO     BY  2000;   /*No help given, try next step  */
%REPLACE FDV$K_UHELPED      BY  2001;   /*Help given, continue sequence */
%REPLACE FDV$K_UHELP_ALL    BY  2002;   /*Help given, repeat UAR        */
/*********************************/
/* Function Key UAR return codes */
/*********************************/
%REPLACE FDV$K_UKEY_ERR     BY  3000;   /*Fn Key failure, FDV signals   */
%REPLACE FDV$K_UKEY_TRM     BY  3001;   /*Fn Key success, normal F.K.   */
%REPLACE FDV$K_UKEY_NXT     BY  3002;   /*Fn Key succ, treat as NEXT    */
%REPLACE FDV$K_UKEY_NTR     BY  3003;   /*Fn Key succ, treat as ENTER   */
%REPLACE FDV$K_UKEY_SUC     BY  3004;   /*Fn Key succ, ignore           */
/*****************************************************************************/
/* FDV status codes returned when FDV$... routines are called as functions. */
/* These codes are VMS status codes and can be signalled.  They correspond  */
/* one-to-one with the FMS status codes retrievable from FDV$STAT.          */
/*****************************************************************************/
%REPLACE FDV$_SUC BY 2719889 ;
%REPLACE FDV$_INC BY 2719897 ;
%REPLACE FDV$_MOD BY 2719905 ;
%REPLACE FDV$_IMP BY 2719922 ;
%REPLACE FDV$_FSP BY 2719930 ;
%REPLACE FDV$_IOL BY 2719938 ;
%REPLACE FDV$_FLB BY 2719946 ;
%REPLACE FDV$_ICH BY 2719954 ;
%REPLACE FDV$_FCH BY 2719962 ;
%REPLACE FDV$_FRM BY 2719970 ;
%REPLACE FDV$_FNM BY 2719978 ;
%REPLACE FDV$_LIN BY 2719986 ;
%REPLACE FDV$_FLD BY 2719994 ;
%REPLACE FDV$_NOF BY 2720002 ;
%REPLACE FDV$_DSP BY 2720010 ;
```

```
%REPLACE FDV$_NSC BY 2720018 ;
%REPLACE FDV$_DNM BY 2720026 ;
%REPLACE FDV$_DLN BY 2720034 ;
%REPLACE FDV$_UTR BY 2720042 ;
%REPLACE FDV$_IOR BY 2720050 ;
%REPLACE FDV$_IFN BY 2720058 ;
%REPLACE FDV$_ARG BY 2720066 ;
%REPLACE FDV$_INI BY 2720074 ;
%REPLACE FDV$_STR BY 2720082 ;
%REPLACE FDV$_IVM BY 2720090 ;
%REPLACE FDV$_FVM BY 2720098 ;
%REPLACE FDV$_ITT BY 2720106 ;
%REPLACE FDV$_TCA BY 2720114 ;
%REPLACE FDV$_STA BY 2720122 ;
%REPLACE FDV$_WID BY 2720130 ;
%REPLACE FDV$_NFL BY 2720138 ;
%REPLACE FDV$_IBF BY 2720146 ;
%REPLACE FDV$_NDS BY 2720154 ;
%REPLACE FDV$_UDP BY 2720162 ;
%REPLACE FDV$_UAR BY 2720170 ;
%REPLACE FDV$_UNF BY 2720178 ;
%REPLACE FDV$_CAN BY 2720194 ;
%REPLACE FDV$_KIF BY 2720202 ;
%REPLACE FDV$_KEX BY 2720210 ;
%REPLACE FDV$_KTW BY 2720218 ;
%REPLACE FDV$_KIL BY 2720226 ;
%REPLACE FDV$_TMO BY 2720234 ;
%REPLACE FDV$_LLI BY 2720242 ;
%REPLACE FDV$_VAL BY 2720250 ;
%REPLACE FDV$_IFU BY 2720258 ;
%REPLACE FDV$_SYS BY 2720266 ;

/*****************************************************************************/
/* FMS status codes returned when FDV$STAT routine is called.                */
/*****************************************************************************/
/* Success codes. */
%REPLACE FDV$K_SUC BY   1;
%REPLACE FDV$K_INC BY   2;
%REPLACE FDV$K_MOD BY   3;

/* Failure code */
%REPLACE FDV$K_IMP BY   -2;
```

```
%REPLACE FDV$K_FSP BY   -3;
%REPLACE FDV$K_IOL BY   -4;
%REPLACE FDV$K_FLB BY   -5;
%REPLACE FDV$K_ICH BY   -6;
%REPLACE FDV$K_FCH BY   -7;
%REPLACE FDV$K_FRM BY   -8;
%REPLACE FDV$K_FNM BY   -9;
%REPLACE FDV$K_LIN BY   -10;
%REPLACE FDV$K_FLD BY   -11;
%REPLACE FDV$K_NOF BY   -12;
%REPLACE FDV$K_DSP BY   -13;
%REPLACE FDV$K_NSC BY   -14;
%REPLACE FDV$K_DNM BY   -15;
%REPLACE FDV$K_DLN BY   -16;
%REPLACE FDV$K_UTR BY   -17;
%REPLACE FDV$K_IOR BY   -18;
%REPLACE FDV$K_IFN BY   -19;
%REPLACE FDV$K_ARG BY   -20;
%REPLACE FDV$K_INI BY   -21;
%REPLACE FDV$K_STR BY   -22;
%REPLACE FDV$K_FVM BY   -23;
%REPLACE FDV$K_IVM BY   -24;
%REPLACE FDV$K_ITT BY   -25;
%REPLACE FDV$K_TCA BY   -26;
%REPLACE FDV$K_STA BY   -27;
%REPLACE FDV$K_WID BY   -28;
%REPLACE FDV$K_NFL BY   -29;
%REPLACE FDV$K_IBF BY   -30;
%REPLACE FDV$K_NDS BY   -31;
%REPLACE FDV$K_UDP BY   -33;
%REPLACE FDV$K_UAR BY   -34;
%REPLACE FDV$K_UNF BY   -35;
%REPLACE FDV$K_CAN BY   -39;
%REPLACE FDV$K_KIF BY   -40;
%REPLACE FDV$K_KEX BY   -41;
%REPLACE FDV$K_KTW BY   -42;
%REPLACE FDV$K_KIL BY   -43;
%REPLACE FDV$K_TMO BY   -44;
%REPLACE FDV$K_LLI BY   -45;
%REPLACE FDV$K_VAL BY   -47;
%REPLACE FDV$K_IFU BY   -48;
%REPLACE FDV$K_SYS BY   -49;
```

```
/*****************************************************/
/* Declare the FDV routines' calling sequence.       */
/* This is the sequence for functions. For calling FDV  */
/* as calls, remove "RETURNS(FIXED BIN)" from each line.*/
/*****************************************************/
DCL FDV$ADLVA   EXTERNAL ENTRY( FIXED BIN )                                                                    RETURNS(FIXED BIN);
DCL FDV$AFVA    EXTERNAL ENTRY( FIXED BIN, CHAR(*), FIXED BIN )                                  OPTIONS(VARIABLE)     RETURNS(FIXED BIN);
DCL FDV$ATERM   EXTERNAL ENTRY( (*)FIXED BIN, FIXED BIN, FIXED BIN, CHAR(*) )                    OPTIONS(VARIABLE)     RETURNS(FIXED BIN);
DCL FDV$AWKSP   EXTERNAL ENTRY( (*)FIXED BIN, FIXED BIN )                                                      RETURNS(FIXED BIN);
DCL FDV$BELL    EXTERNAL ENTRY()                                                                               RETURNS(FIXED BIN);
DCL FDV$CANCL   EXTERNAL ENTRY()                                                                               RETURNS(FIXED BIN);
DCL FDV$CDISP   EXTERNAL ENTRY( CHAR(*), FIXED BIN )                                             OPTIONS(VARIABLE)     RETURNS(FIXED BIN);
DCL FDV$CLEAR   EXTERNAL ENTRY( FIXED BIN, FIXED BIN )                                           OPTIONS(VARIABLE)     RETURNS(FIXED BIN);
DCL FDV$DEL     EXTERNAL ENTRY( CHAR(*) )                                                                      RETURNS(FIXED BIN);
DCL FDV$DFKBD   EXTERNAL ENTRY( (*)FIXED BIN(15), FIXED BIN )                                                  RETURNS(FIXED BIN);
DCL FDV$DISP    EXTERNAL ENTRY( CHAR(*), FIXED BIN )                                             OPTIONS(VARIABLE)     RETURNS(FIXED BIN);
DCL FDV$DISPW   EXTERNAL ENTRY( FIXED BIN )                                                      OPTIONS(VARIABLE)     RETURNS(FIXED BIN);
DCL FDV$DPCOM   EXTERNAL ENTRY( FIXED BIN )                                                                    RETURNS(FIXED BIN);
DCL FDV$DTERM   EXTERNAL ENTRY( (*)FIXED BIN )                                                                 RETURNS(FIXED BIN);
DCL FDV$DWKSP   EXTERNAL ENTRY( (*)FIXED BIN )                                                                 RETURNS(FIXED BIN);
DCL FDV$GET     EXTERNAL ENTRY( CHAR(*), FIXED BIN, CHAR(*), FIXED BIN )                         OPTIONS(VARIABLE)     RETURNS(FIXED BIN);
DCL FDV$GETAF   EXTERNAL ENTRY( CHAR(*), FIXED BIN, CHAR(*), FIXED BIN )                         OPTIONS(VARIABLE)     RETURNS(FIXED BIN);
DCL FDV$GETAL   EXTERNAL ENTRY( CHAR(*), FIXED BIN, CHAR(*), FIXED BIN )                         OPTIONS(VARIABLE)     RETURNS(FIXED BIN);
DCL FDV$GETDL   EXTERNAL ENTRY( CHAR(*), FIXED BIN, FIXED BIN, CHAR(*) )                         OPTIONS(VARIABLE)     RETURNS(FIXED BIN);
DCL FDV$GETSC   EXTERNAL ENTRY( CHAR(*), CHAR(*), FIXED BIN )                                    OPTIONS(VARIABLE)     RETURNS(FIXED BIN);
DCL FDV$ILTRM   EXTERNAL ENTRY( FIXED BIN )                                                                    RETURNS(FIXED BIN);
DCL FDV$LCHAN   EXTERNAL ENTRY( FIXED BIN )                                                                    RETURNS(FIXED BIN);
DCL FDV$LCLOS   EXTERNAL ENTRY()                                                                               RETURNS(FIXED BIN);
DCL FDV$LEDOF   EXTERNAL ENTRY( FIXED BIN )                                                                    RETURNS(FIXED BIN);
DCL FDV$LEDON   EXTERNAL ENTRY( FIXED BIN )                                                                    RETURNS(FIXED BIN);
DCL FDV$LOAD    EXTERNAL ENTRY( CHAR(*) )                                                                      RETURNS(FIXED BIN);
DCL FDV$LOPEN   EXTERNAL ENTRY( CHAR(*), FIXED BIN )                                             OPTIONS(VARIABLE)     RETURNS(FIXED BIN);
DCL FDV$NDISP   EXTERNAL ENTRY()                                                                               RETURNS(FIXED BIN);
DCL FDV$PFT     EXTERNAL ENTRY( FIXED BIN, CHAR(*), CHAR(*), CHAR(*), FIXED BIN ) OPTIONS(VARIABLE)            RETURNS(FIXED BIN);
DCL FDV$PUT     EXTERNAL ENTRY( CHAR(*), CHAR(*), FIXED BIN )                                    OPTIONS(VARIABLE)     RETURNS(FIXED BIN);
DCL FDV$PUTAL   EXTERNAL ENTRY( CHAR(*) )                                                                      RETURNS(FIXED BIN);
DCL FDV$PUTD    EXTERNAL ENTRY( CHAR(*), FIXED BIN )                                             OPTIONS(VARIABLE)     RETURNS(FIXED BIN);
DCL FDV$PUTDA   EXTERNAL ENTRY()                                                                               RETURNS(FIXED BIN);
DCL FDV$PUTL    EXTERNAL ENTRY( CHAR(*), FIXED BIN )                                             OPTIONS(VARIABLE)     RETURNS(FIXED BIN);
DCL FDV$PUTSC   EXTERNAL ENTRY( CHAR(*), CHAR(*) )                                               OPTIONS(VARIABLE)     RETURNS(FIXED BIN);
DCL FDV$READ    EXTERNAL ENTRY( CHAR(*), (*)FIXED BIN, FIXED BIN, FIXED BIN )                                  RETURNS(FIXED BIN);
DCL FDV$RET     EXTERNAL ENTRY( CHAR(*), CHAR(*), FIXED BIN )                                    OPTIONS(VARIABLE)     RETURNS(FIXED BIN);
DCL FDV$RETAL   EXTERNAL ENTRY( CHAR(*) )                                                                      RETURNS(FIXED BIN);
DCL FDV$RETCX   EXTERNAL ENTRY( POINTER, POINTER, CHAR(*), CHAR(*) , FIXED BIN,
                                FIXED BIN, FIXED BIN, FIXED BIN )                                              RETURNS(FIXED BIN);
```

```
DCL FDV$RETDI   EXTERNAL ENTRY( FIXED BIN, CHAR(*), CHAR(*) )                          OPTIONS(VARIABLE)       RETURNS(FIXED BIN);
DCL FDV$RETDN   EXTERNAL ENTRY( CHAR(*) , CHAR(*), FIXED BIN )                        OPTIONS(VARIABLE)       RETURNS(FIXED BIN);
DCL FDV$RETFL   EXTERNAL ENTRY( FIXED BIN, CHAR(*), FIXED BIN, FIXED BIN )            OPTIONS(VARIABLE)       RETURNS(FIXED BIN);
DCL FDV$RETFN   EXTERNAL ENTRY( CHAR(*), FIXED BIN )                                  OPTIONS(VARIABLE)       RETURNS(FIXED BIN);
DCL FDV$RETFO   EXTERNAL ENTRY( FIXED BIN, CHAR(*), FIXED BIN )                                               RETURNS(FIXED BIN);
DCL FDV$RETLE   EXTERNAL ENTRY( FIXED BIN, CHAR(*), FIXED BIN )                        OPTIONS(VARIABLE)       RETURNS(FIXED BIN);
DCL FDV$RFRSH   EXTERNAL ENTRY()                                                                               RETURNS(FIXED BIN);
DCL FDV$SIGOP   EXTERNAL ENTRY()                                                                               RETURNS(FIXED BIN);
DCL FDV$SPADA   EXTERNAL ENTRY( FIXED BIN )                                                                    RETURNS(FIXED BIN);
DCL FDV$SPOFF   EXTERNAL ENTRY()                                                                               RETURNS(FIXED BIN);
DCL FDV$SPON    EXTERNAL ENTRY()                                                                               RETURNS(FIXED BIN);
DCL FDV$SSIGQ   EXTERNAL ENTRY( FIXED BIN )                                                                    RETURNS(FIXED BIN);
DCL FDV$SSRV    EXTERNAL ENTRY( FIXED BIN, FIXED BIN )                                 OPTIONS(VARIABLE)       RETURNS(FIXED BIN);
DCL FDV$STAT    EXTERNAL ENTRY( FIXED BIN, FIXED BIN )                                 OPTIONS(VARIABLE)       RETURNS(FIXED BIN);
DCL FDV$SWKSP   EXTERNAL ENTRY( (*)FIXED BIN )                                         OPTIONS(VARIABLE)       RETURNS(FIXED BIN);
DCL FDV$WAIT    EXTERNAL ENTRY( FIXED BIN )                                            OPTIONS(VARIABLE)       RETURNS(FIXED BIN);
```

Appendix A
VAX–11 FMS Form Driver Calls

A.1 VAX–11 Language-Independent Notation

Form Driver routines are invoked according to rules specified in the VAX–11 Procedure Calling and Condition Handling Standard (Appendix C of the *VAX–11 Run-Time Library Reference Manual*). The complete notation for describing VAX–11 calls is documented in Appendix C of the *VAX–11 Guide to Creating Modular Library Procedures.*

Form Driver routines can be invoked as subroutines or as functions:

As a subroutine **CALL FDV$xxx (parameter1,parameter2,...)**

As a function **VMS_stat.wlc.v = FDV$xxx (parameter1,parameter2,...)**

The access type, data type, passing mechanism, and parameter form are described in the following prescribed order:

<parameter-name>.<access type><data type>.<passing mechanism><parameter form>

Example

For the FDV$GET call the **fldval**, **fldtrm**, **fldnam**, and **fldidx** parameters are described as follows:

FDV$GET (fldval.wt.dx1,fldtrm.wl.r,fldnam.rt.dx1[,fldidx.rl.r]

The notation for each parameter is explained below. Note that every Form Driver call returns a VMS status code in the form **VMS_stat.wlc.v.**

Parameter	<access type>	<data type>	<passing mechanism>	<parameter form>
fldval	**w** Write-only access	**t** Character-coded text string	**d** By descriptor	**x1** Fixed-length or dynamic string descriptor
fldtrm	**w** Write-only access	**l** Longword integer (signed)	**r** By reference	—
fldnam	**r** Read-only access	**t** Character-coded text string	**d** By descriptor	**x1** Fixed-length or dynamic string descriptor
fldidx	**r** Read-only access	**l** Longword integer (signed)	**r** By reference	—

A.2 Procedure Parameter Notation for Form Driver Calls

FMS uses a subset of the VAX–11 procedure parameter notation. The following table explains the notation used for access type, data type, passing mechanism, and parameter form.

Notation	<access type>	Comments
m	Modify access	Parameters for both input and output
r	Read-only access	Parameters for input
w	Write-only access	Parameters for output

Notation	<data type>
a	Virtual address
l	Longword integer (signed)
lc	Longword return status
t	Character-coded text string
v	Aligned bit string
w	Word integer (signed)

Notation	<passing mechanism>	Comments
d	By descriptor	FMS passing mechanism for character strings and integer arrays
r	By reference	FMS passing mechanism for integers

Notation	<parameter form>
a	Array reference or descriptor
x1	Fixed-length or dynamic string descriptor

Call	Procedure Parameter Notation
ADLVA	**FDV$ADLVA (video.ml.r)** **video** video attributes code of data line Alters the data line video attributes. You can specify Blink, Bold, Reverse, and/or Underline.
AFCX	**FDV$AFCX (insovr.rl.r,curpos.rl.r[,fldnam.rt.dx1[,fldidx.rl.r]])** **insovr** Insert/Overstrike mode of field **curpos** cursor position within field **fldnam** field name **fldidx** field index Alters the default field context of the specified field. You can specify Insert or Overstrike mode and cursor position in the field before any GET operation involving that field.
AFVA	**FDV$AFVA (video.ml.r[,fldnam.rt.dx1[,fldidx.rl.r]])** **video** video attributes code for field **fldnam** field name **fldidx** field index Alters the field video attributes.
ATERM	**FDV$ATERM (tca.ml.da,size.rl.r,channel.rl.r[,trmnal.rt.dx1])** or **FDV$ATERM (tca.mt.dx1,size.rl.r,channel.rl.r[,trmnal.rt.dx1])** **tca** terminal control area **size** terminal control area size **channel** logical I/O channel number for terminal **trmnal** name of terminal Attaches a terminal to the Form Driver for processing forms over a specific, logical I/O channel, names a TCA for that terminal, and specifies the size of the TCA.
AWKSP	**FDV$AWKSP (wksp.ml.da,size.rl.r)** or **FDV$AWKSP (wksp.mt.dx1,size.rl.r)** **wksp** form workspace **size** estimate of workspace size Attaches a form workspace to a list of workspaces associated with the current TCA, specifies the size in bytes, and establishes that workspace as the current workspace.
BELL	**FDV$BELL** Rings the terminal bell.
CANCL	**FDV$CANCL** Cancels any other call presently being processed on the current terminal.

(continued on next page)

Call	Procedure Parameter Notation
CDISP	**FDV$CDISP (frmnam.rt.dx1[,offset.rl.r])** **frmnam** form name **offset** number controlling placement of form on screen Clears the screen and displays a form. The display position may be offset from the original form description.
CLEAR	**FDV$CLEAR ([line[,linecnt]])** **line** line number of first line to clear **linecnt** number of lines to clear Clears the entire screen unless otherwise specified with the arguments.
DEL	**FDV$DEL (frmnam.rt.dx1)** **frmnam** form name Deletes a form from the list of memory-resident forms.
DFKBD	**FDV$DFKBD (defkbd.rw.da,kbdnum.rl.r)** **defkbd** array of key functions and key codes **kbdnum** number of pairs of key functions and associated key codes in **defkbd** array Redefines the FMS keypad function keys.
DISP	**FDV$DISP (frmnam.rt.dx1[,offset.rl.r])** **frmnam** form name **offset** number controlling placement of form on screen Clears the portion of the screen specified as the "clear area" in the form description and displays a form. The display position can be offset from the original form description.
DISPW	**FDV$DISPW ([offset.rl.r])** **offset** number controlling placement of form on screen Clears the portion of the screen specified as the "clear area" in the form description and displays the form that is already loaded in the workspace. The display position can be offset from the original form description.
DPCOM	**FDV$DPCOM ([dpmode.rl.r])** **dpmode** value defining decimal point in signed-numeric fields Defines the comma, or redefines the period, as the decimal point in fields containing signed-numeric field-validation characters.
DTERM	**FDV$DTERM (tca.ml.da)** or **FDV$DTERM (tca.mt.dx1)** **tca** terminal control area Clears the terminal screen, detaches a terminal from the Form Driver, and detaches any workspaces associated with the terminal.

(continued on next page)

Call	Procedure Parameter Notation
DWKSP	**FDV$DWKSP (wksp.ml.da)** or **FDV$DWKSP (wksp.rt.dx1)** **wksp** form workspace Detaches a form workspace from the list of workspaces associated with the current terminal.
GET	**FDV$GET (fldval.wt.dx1,fldtrm.wl.r,fldnam.rt.dx1[,fldidx.rl.r])** **fldval** field value **fldtrm** field terminator **fldnam** field name **fldidx** field index Positions the cursor in the initial cursor position of a specific modifiable field and waits for the operator to enter a value.
GETAF	**FDV$GETAF (fldval.wt.dx1,fldtrm.wl.r,fldnam.wt.dx1[,fldidx.wl.r])** **fldval** field value **fldtrm** field terminator **fldnam** ending field name **fldidx** ending field index Positions the cursor in the current field in the form and waits for the operator to enter a value in any field.
GETAL	**FDV$GETAL ([fldval.wt.dx1,fldtrm.wl.r[,fldnam.rt.dx1[,fldidx.rl.r]]])** **fldval** returned values of all fields in form **fldtrm** field terminator **fldnam** starting field name **fldidx** starting field index Positions the cursor in the first modifiable field in a form unless otherwise specified in the **fldnam** argument and allows you to enter data in all modifiable, nonscrolled fields.
GETDL	**FDV$GETDL (value.wt.dx1,fldtrm.wl.r[,line.rl.r[,prompt.rt.dx1]])** **value** contents of data line returned from Form Driver **fldtrm** field terminator **line** line number on which the operator's input is displayed **prompt** data line text to serve as a prompt for the operator Gets a data line from a specified line on the screen.
GETSC	**FDV$GETSC (fldnam.rt.dx1,fldval.wt.dx1[,fldtrm.wl.r])** **fldnam** field name that identifies a scrolled area **fldval** field values **fldtrm** field terminator Positions the cursor within the current line in the scrolled area that contains the specified field and accepts input in modifiable fields within the line.

(continued on next page)

Call	Procedure Parameter Notation
ILTRM	**FDV$ILTRM ([trmmod.rl.r])**
	trmmod value for illegal terminator mode switch
	Specifies the action to take when an illegal field terminator is entered. An illegal field terminator can be rejected by the Form Driver or returned to the program.
LCHAN	**FDV$LCHAN (channel.rl.r)**
	channel I/O channel number for form library
	Sets the channel for form library files associated with the current terminal. The Form Driver uses the specified channel for any LOPEN or LCLOS call processing.
LCLOS	**FDV$LCLOS**
	Closes the form library associated with the current library channel for the current terminal.
LEDOF	**FDV$LEDOF (ledno.rl.r)**
	ledno terminal LED number
	Turns off the light-emitting diode (LED) on the VT100 keyboard.
LEDON	**FDV$LEDON (ledno.rl.r)**
	ledno terminal LED number
	Turns on the light-emitting diode (LED) on the VT100 keyboard.
LOAD	**FDV$LOAD (frmnam.rt.dx1)**
	frmnam form name
	Loads a form description into a workspace without displaying the form on the screen.
LOPEN	**FDV$LOPEN (filspc.rt.dx1[,channel.rl.r])**
	filspc form library file specification **channel** I/O channel number for form library
	Opens a form library and replaces the current library channel specification if the I/O channel number is supplied.
NDISP	**FDV$NDISP**
	Marks current workspace as not displayed.

(continued on next page)

Call	Procedure Parameter Notation
PFT	**FDV$PFT ([fldtrm.rl.r[,fldnam.rt.dx1[,fldval.rt.dx1 [,nfldnam.wt.dx1[,nfldidx.wl.r]]]]])**
	fldtrm field terminator to be processed **fldnam** field name that identifies a scrolled area **fldval** field values to be displayed **nfldnam** current field name after call has been completed **nfldidx** current field index after call has been completed
	Processes the field terminator and checks for valid terminator codes.
PUT	**FDV$PUT (fldval.rt.dx1,fldnam.rt.dx1[,fldidx.rl.r])**
	fldval field value to be displayed **fldnam** field name **fldidx** field index
	Stores the value of the **fldval** argument and displays that value in the specified field.
PUTAL	**FDV$PUTAL (frmval.rt.dx1)**
	frmval list of field values to be displayed
	Outputs values to all fields, stores the **frmval** argument values in the workspace for nonscrolled fields, and displays these values on the screen.
PUTD	**FDV$PUTD (fldnam.rt.dx1[,fldidx.rl.r])**
	fldnam field name **fldidx** field index
	Outputs the default value to a specified field.
PUTDA	**FDV$PUTDA**
	Outputs default values to all fields in the form and displays those values on the screen.
PUTL	**FDV$PUTL (text.rt.dx1[,line.rl.r])**
	text data line text **line** line number for displayed data line
	Outputs data to the specified line on the screen. If the line number is zero, the data line is displayed on the last line of the screen.
PUTSC	**FDV$PUTSC (fldnam.rt.dx1[,fldval.rt.dx1])**
	fldnam field name that identifies a scrolled area **fldval** field value
	Outputs data to the current line of a scrolled area that contains the specified field name.

(continued on next page)

Call	Procedure Parameter Notation
READ	**FDV$READ (frmnam.rt.dx1,mloc.ml.da,mlocsiz.rl.r,frmsiz.wl.r)** or **FDV$READ (frmnam.rt.dx1,mloc.rt.dx1,mlocsiz.rl.r,frmsiz.wl.r)** **frmnam** form name **mloc** form storage area **mlocsiz** size of memory buffer that begins with **mloc** **frmsiz** form size actually used Extracts a form from the current form library, stores it in a specified memory area, and adds the name of the form to the list of memory-resident forms.
RET	**FDV$RET (fldval.wt.dx1,fldnam.rt.dx1[,fldidx.rl.r])** **fldval** field value **fldnam** field name **fldidx** field index Returns the value for a specified field stored in the workspace.
RETAL	**FDV$RETAL (frmval.wt.dx1)** **frmval** concatenated values of all fields except those in scrolled areas Returns the values for all fields except those in scrolled areas stored in the workspace.
RETCX	**FDV$RETCX (atca.wa.r,awksp.wa.r,frmnam.wt.dx1,uarval.wt.dx1, curpos.wl.r,fldtrm.wl.r,insovr.wl.r,hlpnum.wl.r)** **atca** terminal control area address **awksp** form workspace address **frmnam** form name **uarval** value of the associated text for this UAR **curpos** cursor position within field **fldtrm** returned field terminator **insovr** Insert/Overstrike mode of field **hlpnum** number of times HELP key pressed for current field Returns the current context of the Form Driver. You can issue this call in a UAR to determine the context in which the UAR is called.
RETDI	**FDV$RETDI (nmdidx.rl.r,nmdval.wt.dx1[,nmdnam.wt.dx1])** **nmdidx** index of Named Data item **nmdval** text of Named Data item **nmdnam** name of Named Data item Returns the Named Data text that you specify by its index (rather than by its name).

(continued on next page)

Call	Procedure Parameter Notation
RETDN	**FDV$RETDN (nmdnam.rt.dx1,nmdval.wt.dx1[,nmdidx.rl.r])**
	nmdnam name of Named Data item **nmdval** text of Named Data item **nmdidx** index of Named Data item
	Returns the Named Data text that you specify by its name (rather than by its index).
RETFL	**FDV$RETFL (line.rl.r,value.wt.dx1,linlen.wl.r[,type.rl.r])**
	line line number of form to be returned **value** value of requested line **linlen** length of character string returned in value parameter **type** type of output line requested
	Returns the contents of the line that you specify with the **line** argument. This is one of the lines displayed by the RFRSH call. This call can also be used for loaded but undisplayed forms for report formatting.
RETFN	**FDV$RETFN (fldnam.wt.dx1[,fldidx.wl.r])**
	fldnam field name **fldidx** field index
	Returns the current field name and index from the current workspace. If the field is not indexed, the index value returned is zero.
RETFO	**FDV$RETFO (fldnum.rl.r,fldnam.wt.dx1,fldidx.wl.r)**
	fldnum field number **fldnam** field name corresponding to **fldnum** **fldidx** field index corresponding to **fldnum**
	Returns the name and index of the nth field in the form.
RETLE	**FDV$RETLE (fldlen.wl.r,fldnam.rt.dx1[,fldidx.rl.r])**
	fldlen field length excluding field-marker characters **fldnam** field name **fldidx** field index
	Returns the number of data characters in the specified field.
RFRSH	**FDV$RFRSH**
	Refreshes the screen. The RFRSH operation is identical to that initiated by pressing the CTRL/R keys.
SIGOP	**FDV$SIGOP**
	Causes the application program to signal the operator.
SPADA	**FDV$SPADA (mode.rl.r)**
	mode numeric/application keypad mode
	Sets the keypad to numeric or application mode.

(continued on next page)

Call	Procedure Parameter Notation
SPOFF	**FDV$SPOFF** Turns supervisor-only mode off for the current terminal, allowing the operator to modify fields protected with the Supervisor Only attribute.
SPON	**FDV$SPON** Turns supervisor-only mode on for the current terminal, treating fields protected with the Supervisor-Only attribute as display-only fields.
SSIGQ	**FDV$SSIGQ (sigmd.rl.r)** **sigmd** bell/reverse video signaling mode Sets signal mode for the current terminal. Audio mode (0) rings the terminal bell. Video mode (1) reverses the VT100 video image.
SSRV	**FDV$SSRV ([status.wl.r[,iostat.wl.r]])** **status** general status reporting variable **iostat** I/O status reporting variable Sets the addresses of the status reporting variables.
STAT	**FDV$STAT (status.wl.r[,iostat.wl.r])** **status** general status code **iostat** I/O status code Returns the status code for the last Form Driver call.
STERM	**FDV$STERM (tca.ml.da)** or **FDV$STERM (tca.rt.dx1)** **tca** terminal control area Sets current terminal and the workspace most recently associated with that terminal to the current workspace. The TCA must have been previously attached by the FDV$ATERM call.
STIME	**FDV$STIME (time.rl.r)** **time** timeout period in seconds Specifies the number of seconds the Form Driver waits for operator response to a GET-type call.
SWKSP	**FDV$SWKSP (wksp.ml.da)** or **FDV$SWKSP (wksp.mt.dx1)** **wksp** form workspace Specifies the workspace that the Form Driver uses for the current workspace. The workspace must have been previously attached by the FDV$ATERM call.

(continued on next page)

VAX–11 FMS Form Driver Calls

Call	Procedure Parameter Notation
TCHAN	**FDV$TCHAN (channel.rl.r)**
	channel physical I/O channel number for terminal
	Changes the terminal channel associated with the current TCA to the specified value.
WAIT	**FDV$WAIT ([fldtrm.wl.r])**
	fldtrm field terminator code
	Causes the application program to wait until the operator presses the ENTER key. This call allows the Form Driver to set the application program to the operator's pace.

Appendix B
Sample Application Program Form Descriptions

The FMS Sample Application program uses thirteen forms. The form descriptions and their screen images are presented in this appendix. The descriptions are written in the Form Language. The *VAX–11 FMS Utilities Reference Manual* describes the Form Language in detail.

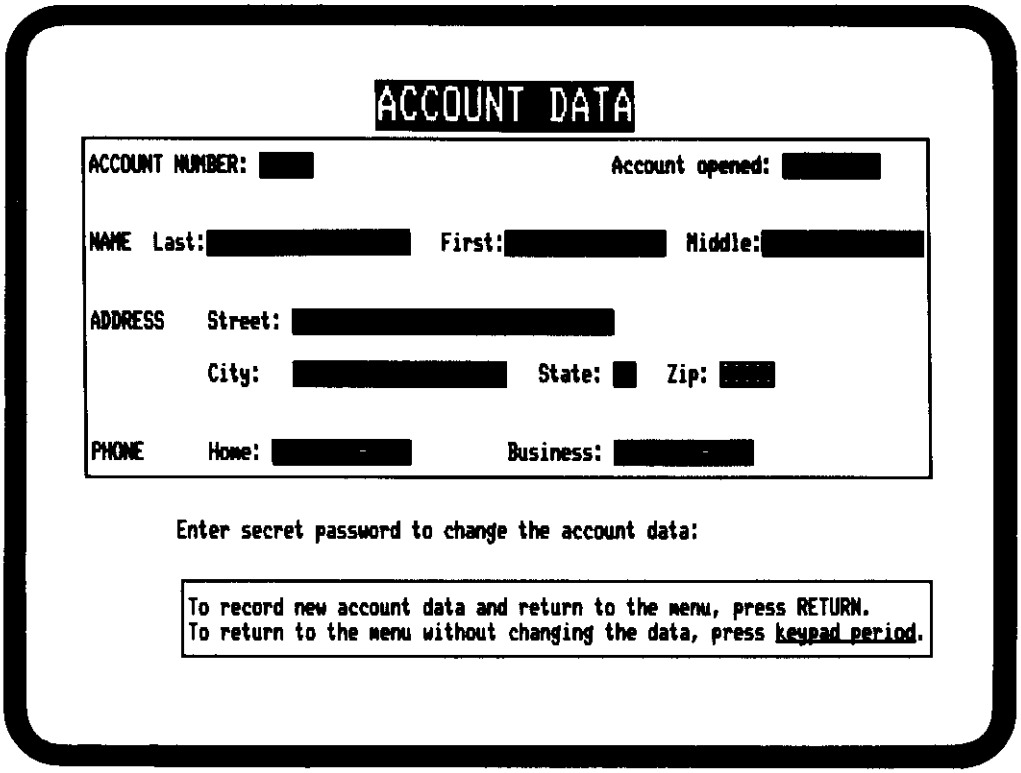
Understanding the forms can help you understand the sample program. Refer to the form descriptions and their screen images as you review the Sample Application program. Some of the screen images in this appendix are not equivalent to their form description, because the images include data supplied by the Sample Application program.

The form descriptions and their screen images appear in the following order:

```
ACCOUNT_DATA
CHECK
CHECK_DONE
DEPOSIT
HELP_ACCOUNT_DATA
HELP_CHECK
HELP_DEPOSIT
HELP_KEYS
HELP_MENU
HELP_WELCOME
MENU
REGISTER
WELCOME
```

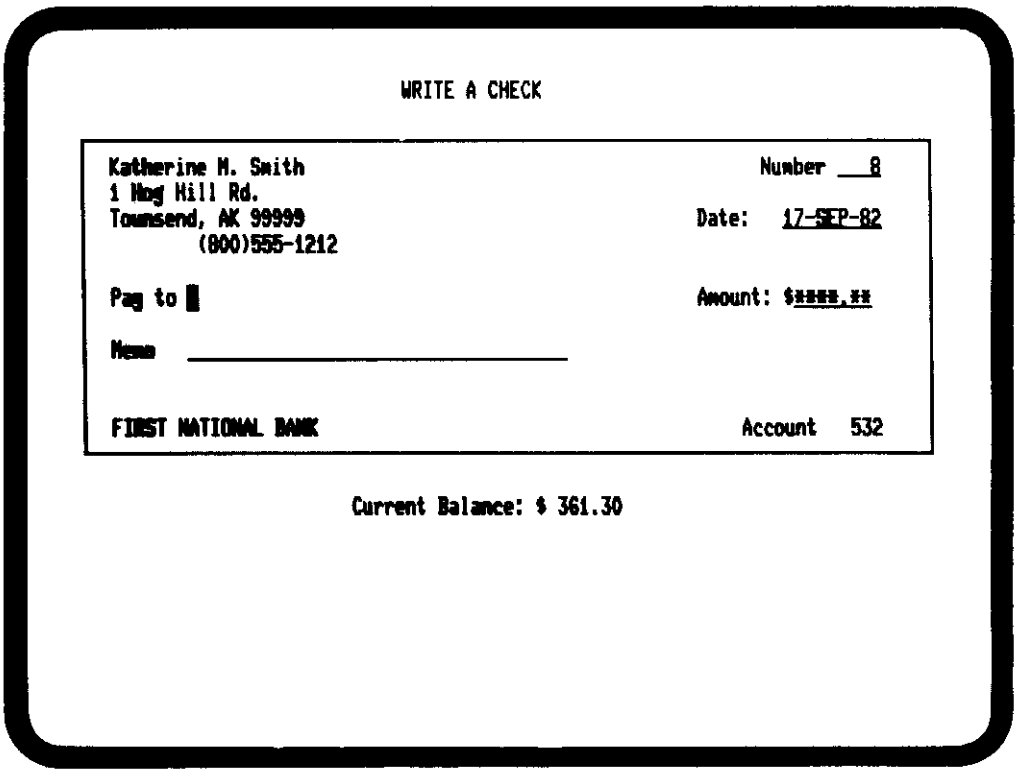
```
!*********************************************************!
!       ACCOUNT_DATA                                      !
!       FMS V2 SAMPLE APPLICATION PROGRAM FORM            !
!       Account data display and entry form               !
!*********************************************************!

FORM NAME=                          'ACCOUNT_DATA'
    HELP_FORM=                      'HELP_ACCOUNT_DATA'
    AREA_TO_CLEAR=                  1:23
    WIDTH=                          80
    BACKGROUND=                     WHITE
    HIGHLIGHT=                      BOLD
    DBLSIZ=                         1
    FUNCTION_KEY_ACTION_ROUTINE=  'PASSKY' : '110' ;


TEXT                        (1,15)   'ACCOUNT DATA' BOLD ;

DRAW                        (3,1) : (16,80);

TEXT                        (4,2)    'ACCOUNT NUMBER:' ;
FIELD NAME= 'ACCTNO'        (4,18)   PICTURE= 5'9'
                                     RIGHT_JUSTIFIED
                                     DISPLAY_ONLY
                                     REVERSE ;

TEXT                        (4,51)   'Account opened:' ;
FIELD NAME= 'DATE'          (4,67)   PICTURE= '99-AAA-99'
                                     DISPLAY_ONLY
                                     REVERSE ;

TEXT                        (7,2)    'NAME  Last:' ;
FIELD NAME= 'LAST'          (7,13)   PICTURE= 20'X'
                                     RESPONSE_REQUIRED
                                     SUPERVISOR_ONLY
                                     REVERSE ;

TEXT                        (7,35)   'First:' ;
FIELD NAME= 'FIRST'         (7,41)   PICTURE= 15'X'
                                     RESPONSE_REQUIRED
                                     SUPERVISOR_ONLY
                                     REVERSE ;

TEXT                        (7,58)   'Middle:' ;
FIELD NAME= 'MIDDLE'        (7,65)   PICTURE= 15'X'
                                     SUPERVISOR_ONLY
                                     REVERSE ;

TEXT                        (10,2)   'ADDRESS    Street:' ;
FIELD NAME= 'STREET'        (10,21)  PICTURE= 30'X'
                                     RESPONSE_REQUIRED
                                     SUPERVISOR_ONLY
                                     REVERSE ;

TEXT                        (12,13)  'City:' ;
FIELD NAME= 'CITY'          (12,21)  PICTURE= 20'X'
                                     RESPONSE_REQUIRED
                                     SUPERVISOR_ONLY
                                     REVERSE ;


TEXT                        (12,44)  'State:' ;
FIELD NAME= 'STATE'         (12,51)  PICTURE= 2'X'
                                     RESPONSE_REQUIRED
                                     SUPERVISOR_ONLY
                                     REVERSE ;
```

```
TEXT                    (12,56) 'Zip:' ;
FIELD NAME= 'ZIP'       (12,61) PICTURE= 5'9'
                                RIGHT_JUSTIFIED
                                ZERO_FILL
                                RESPONSE_REQUIRED
                                SUPERVISOR_ONLY
                                CLEAR_CHARACTER= '0'
                                REVERSE ;

TEXT                    (15,2)  'PHONE      Home:' ;
FIELD NAME= 'HOMEPH'    (15,19) PICTURE= '(999)999-9999'
                                SUPERVISOR_ONLY
                                REVERSE ;

TEXT                    (15,41) 'Business:' ;
FIELD NAME= 'WORKPH'    (15,51) PICTURE= '(999)999-9999'
                                SUPERVISOR_ONLY
                                REVERSE ;

TEXT                    (18,10) 'Enter secret password to change the account '
                                &'data:' ;
FIELD NAME= 'SECRET'    (18,60) PICTURE= 12'X'
                                HELP= 'Secret password: SAMP'
                                NOECHO ;

DRAW                    (20,10) : (23,80) ;

TEXT                    (21,11) 'To record new account data and return to '
                                &'the menu, press RETURN.' ;

TEXT                    (22,11) 'To return to the menu without changing the '
                                &'data, press' ;

TEXT                    (22,66) 'Keypad period'
                                UNDERLINE ;

TEXT                    (22,79) '.' ;

! This form is read with a FDV$GETAL so SAMP expects the following order of
! fields returned.  The fields can be rearranged on the form without changing
! the program as long as this ORDER statement is used.

ORDER BEGIN_WITH =  1
    NAME= 'ACCTNO'
    NAME= 'DATE'
    NAME= 'LAST'
    NAME= 'FIRST'
    NAME= 'MIDDLE'
    NAME= 'STREET'
    NAME= 'CITY'
    NAME= 'STATE'
    NAME= 'ZIP'
    NAME= 'HOMEPH'
    NAME= 'WORKPH'
    NAME= 'SECRET' ;

END_OF_FORM NAME= 'ACCOUNT_DATA' ;
```


ACCOUNT DATA
ACCOUNT NUMBER:
Account opened:
NAME Last:
First:
Middle:
ADDRESS Street:
City:
State:
Zip:
PHONE Home:
Business:
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.

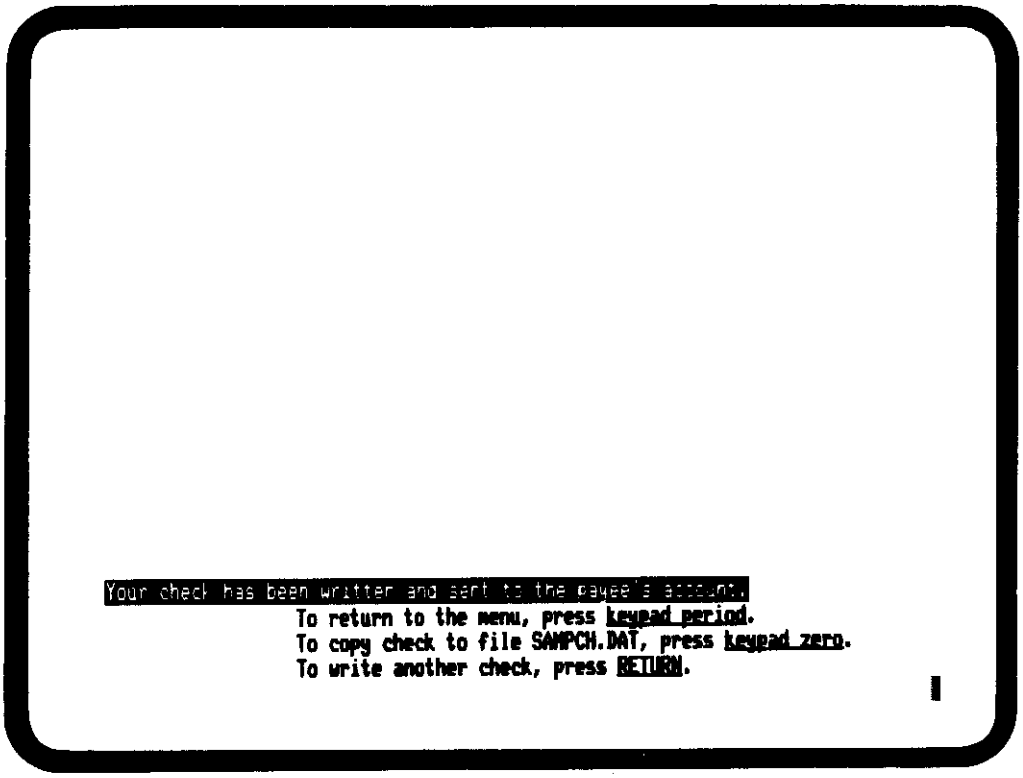
```
!*********************************************************!
!       CHECK                                             !
!       FMS V2 SAMPLE APPLICATION PROGRAM FORM            !
!       Check entry form                                  !
!*********************************************************!

FORM NAME=                          'CHECK'
    HELP_FORM=                      'HELP_CHECK'
    AREA_TO_CLEAR=                  1:23
    WIDTH=                          80
    BACKGROUND=                     CURRENT
    HIGHLIGHT=                      BOLD
    FUNCTION_KEY_ACTION_ROUTINE= 'PASSKY' : '110' ;


TEXT                        (1,31)  'WRITE A CHECK'
                                    BOLD ;

DRAW                        (3,1) : (15,79) ;

FIELD NAME= 'NAME'          (4,4)   PICTURE= 39'X'
                                    DISPLAY_ONLY ;

TEXT                        (4,64)  'Number' ;
FIELD NAME= 'NUMBER'        (4,71)  PICTURE= 4'9'
                                    DISPLAY_ONLY
                                    RIGHT_JUSTIFIED
                                    UNDERLINE ;

FIELD NAME= 'STREET'        (5,4)   PICTURE= 30'X'
                                    DISPLAY_ONLY ;

FIELD NAME= 'CSZ'           (6,4)   PICTURE= 30'X'
                                    DISPLAY_ONLY ;

TEXT                        (6,58)  'Date:' ;
FIELD NAME= 'DATE'          (6,66)  DATE_FIELD= '99-AAA-99'
                                    DISPLAY_ONLY
                                    UNDERLINE ;

FIELD NAME= 'HOMEPH'        (7,12)  PICTURE= '(999)999-9999'
                                    DISPLAY_ONLY ;

TEXT                        (9,4)   'Pay to' ;
FIELD NAME= 'PAYTO'         (9,11)  PICTURE= 35'X'
                                    RESPONSE_REQUIRED
                                    UNDERLINE
        HELP= 'The person/company who is the recipient of your beneficence' ;

TEXT                        (9,58)  'Amount: $' ;
FIELD NAME= 'AMTPAY'        (9,67)  PICTURE= '9999.99'
                                    RIGHT_JUSTIFIED
                                    RESPONSE_REQUIRED
                                    CLEAR_CHARACTER= '*'
                                    UNDERLINE
                                    HELP= 'Enter amount of check'
                                    ACTION_ROUTINE= 'CHKCHK'
                                    ACTION_ROUTINE= 'RANGE'
            :'100, This bank doesn''t issue such small checks. Send cash.' ;

TEXT                        (11,4)  'Memo' ;
FIELD NAME= 'MEMO'          (11,11) PICTURE= 35'X'
                                    UNDERLINE
        HELP= '(Optional) A reminder of why you and your money are parting' ;

TEXT                        (14,4)  'FIRST NATIONAL BANK' ;
```

```
TEXT                        (14,62) 'Account' ;
FIELD NAME= 'ACCTNO'        (14,70) PICTURE= 5'X'
                                    RIGHT_JUSTIFIED
                                    DISPLAY_ONLY ;

TEXT                        (17,26) 'Current Balance: $' ;
FIELD NAME= 'BALANCE'       (17,44) PICTURE= '9999.99'
                                    RIGHT_JUSTIFIED
                                    SUPPRESS ZERO_FILL CLEAR_CHARACTER= '0'
                                    DISPLAY_ONLY ;

! Named Data describes the first and last lines used for check image
! so that the program can generate hard copy of the image.

NAMED_DATA INDEX= 1 NAME= 'FIRST' DATA= '03' ;
NAMED_DATA INDEX= 2 NAME= 'LAST'  DATA= '15' ;

END_OF_FORM NAME= 'CHECK' ;
```

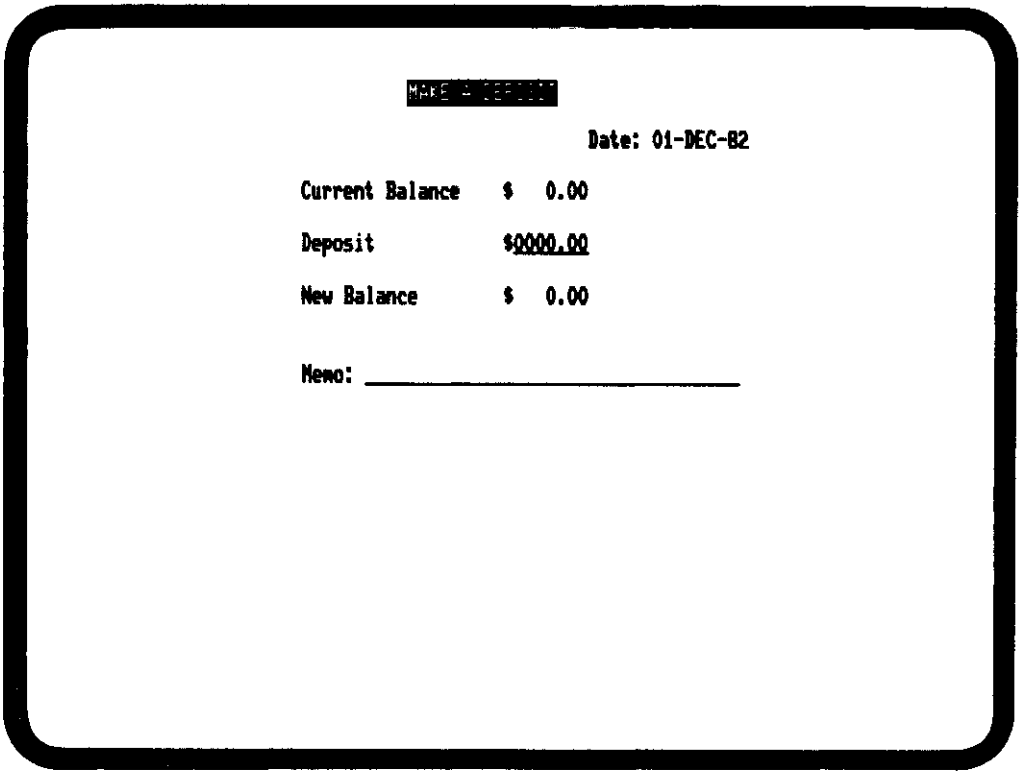


This form shows data supplied by the Sample Application program.

```
!*****************************************************!
!       CHECK_DONE                                    !
!       FMS V2 SAMPLE APPLICATION PROGRAM FORM        !
!       Ask for next action after a check is done.    !
!*****************************************************!

FORM NAME=                          'CHECK_DONE'
    HELP_FORM=                      'HELP_CHECK'
    AREA_TO_CLEAR=                  20:23
    WIDTH=                          CURRENT
    BACKGROUND=                     CURRENT
    CHARACTER_SET=                  US
    FUNCTION_KEY_ACTION_ROUTINE= 'PASSKY' : '110 112   ' ;


TEXT (20,3)       'Your check has been written and sent to the payee''s account.'
                  BOLD ;

TEXT (21,21)      'To return to the menu, press' ;
TEXT (21,50)      'Keypad period'
                  UNDERLINE ;
TEXT (21,63)      '.' ;

TEXT (22,21)      'To copy check to file SAMPCH.DAT, press' ;
TEXT (22,61)      'Keypad zero'
                  UNDERLINE ;
TEXT (22,72)      '.' ;

TEXT (23,21)      'To write another check, press' ;
TEXT (23,51)      'RETURN'
                  UNDERLINE ;
TEXT (23,57)      '.' ;

END_OF_FORM NAME= 'CHECK_DONE' ;
```

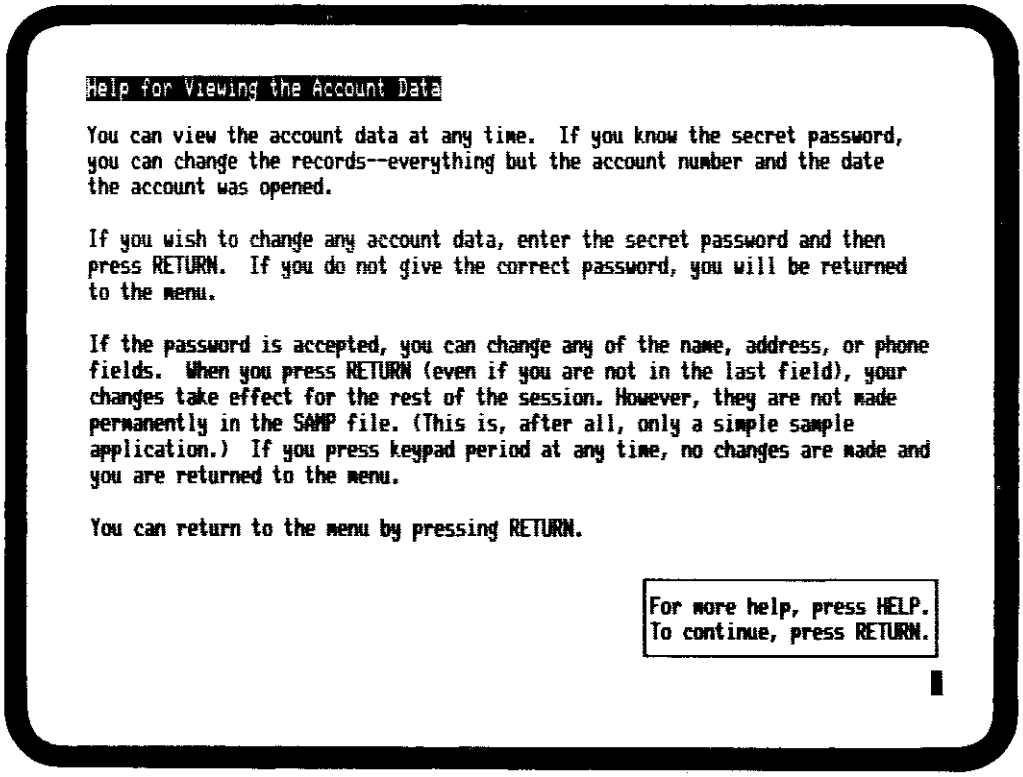


```
!*****************************************************!
!       DEPOSIT                                       !
!       FMS V2 SAMPLE APPLICATION PROGRAM FORM        !
!       Checking account deposit form                 !
!*****************************************************!

FORM NAME=                          'DEPOSIT'
    HELP_FORM=                      'HELP_DEPOSIT'
    AREA_TO_CLEAR=                  1:23
    WIDTH=                          80
    BACKGROUND=                     CURRENT
    FUNCTION_KEY_ACTION_ROUTINE='PASSKY' : '110' ;

TEXT                        (1,32)  'MAKE A DEPOSIT'
                                    BOLD ;

TEXT                        (3,49)  'Date:' ;
FIELD NAME= 'DATE'          (3,55)  DATE_FIELD= '99-AAA-99'
                                    DISPLAY_ONLY  ;

TEXT                        (5,22)  'Current Balance    $' ;
FIELD NAME= 'CURBAL'        (5,42)  PICTURE= '9999.99'
                                    RIGHT_JUSTIFIED
                                    SUPPRESS ZERO_FILL CLEAR_CHARACTER= '0'
                                    DISPLAY_ONLY  ;

TEXT                        (7,22)  'Deposit            $' ;
FIELD NAME= 'DEPOSIT'       (7,42)  PICTURE= '9999.99'
                                    HELP= 'Enter amount of deposit'
                                    FIXED_DECIMAL
                                    ZERO_FILL
                                    RESPONSE_REQUIRED
                                    CLEAR_CHARACTER= '0'
                                    UNDERLINE  ;

TEXT                        (9,22)  'New Balance        $' ;
FIELD NAME= 'NEWBAL'        (9,42)  PICTURE= '9999.99'
                                    RIGHT_JUSTIFIED
                                    SUPPRESS ZERO_FILL CLEAR_CHARACTER= '0'
                                    DISPLAY_ONLY  ;

TEXT                        (12,22) 'Memo:' ;
FIELD NAME= 'MEMO'          (12,28) PICTURE= 35'X'
                                    HELP= 'Enter the origin of the deposit'
                                    RESPONSE_REQUIRED
                                    UNDERLINE  ;

! SAMP gets data from this form via FDV$GETAL.  Define the order expected
! so that the form can be rearranged without changing the program.

ORDER BEGIN_WITH =  1
     NAME= 'DATE'
     NAME= 'CURBAL'
     NAME= 'DEPOSIT'
     NAME= 'NEWBAL'
     NAME= 'MEMO'  ;

! Named Data: used to store a message output by the application.  Storing it
! with the form means that all the interaction text is in one place.  This
! is useful for editing purposes and for language conversions.

NAMED_DATA INDEX= 1 NAME= 'DONE'
    DATA= 'Deposit made. Press RETURN or ENTER to continue.' ;

END_OF_FORM NAME= 'DEPOSIT' ;
```

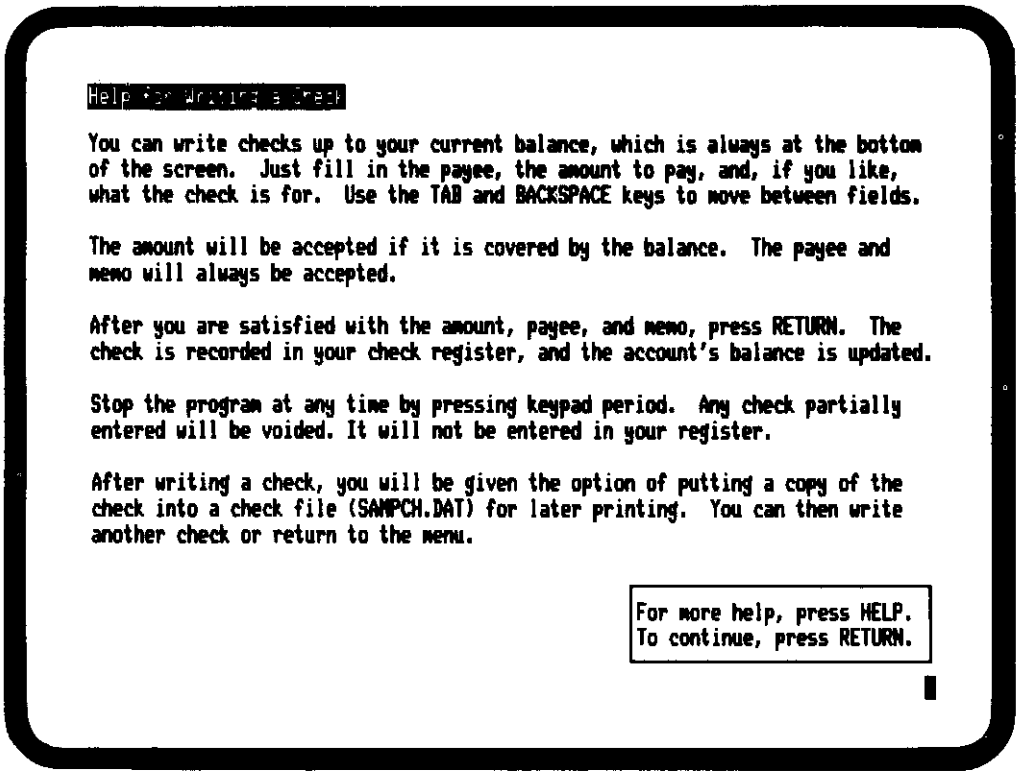


This form shows data supplied by the Sample Application program.

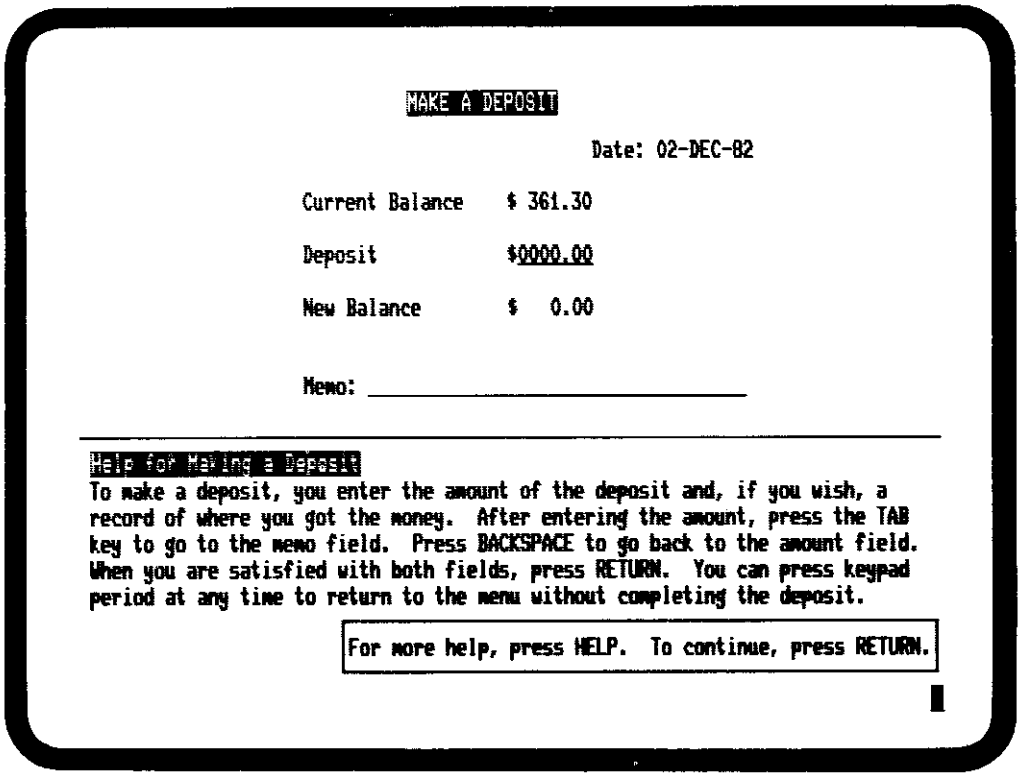
```
!*****************************************************!
!       HELP_ACCOUNT_DATA                             !
!       FMS V2 SAMPLE APPLICATION PROGRAM FORM        !
!       Help form for ACCOUNT_DATA form               !
!*****************************************************!

FORM NAME=                      'HELP_ACCOUNT_DATA'
    HELP_FORM=                  'HELP_KEYS'
    AREA_TO_CLEAR=              1:23
    WIDTH=                      80
    BACKGROUND=                 CURRENT ;


TEXT (1,2)      'Help for Viewing the Account Data'
                BOLD  ;
TEXT (3,2)      'You can view the account data at any time.  If you know the '
                &'secret password,' ;
TEXT (4,2)      'you can change the records--everything but the account number'
                &' and the date' ;
TEXT (5,2)      'the account was opened.' ;
TEXT (7,2)      'If you wish to change any account data, enter the secret '
                &'password and then' ;
TEXT (8,2)      'press RETURN.  If you do not give the correct password, you '
                &'will be returned' ;
TEXT (9,2)      'to the menu.' ;
TEXT (11,2)     'If the password is accepted, you can change any of the name,'
                &' address, or phone' ;
TEXT (12,2)     'fields.  When you press RETURN (even if you are not in the '
                &'last field), your' ;
TEXT (13,2)     'changes take effect for the rest of the session. However, '
                &'they are not made' ;
TEXT (14,2)     'permanently in the SAMP file. (This is, after all, only a '
                &'simple sample' ;
TEXT (15,2)     'application.)  If you press keypad period at any time, no '
                &'changes are made and' ;
TEXT (16,2)     'you are returned to the menu.' ;
TEXT (18,2)     'You can return to the menu by pressing RETURN.' ;

DRAW (20,53) : (23,80);

TEXT (21,54) 'For more help, press HELP.' ;
TEXT (22,54) 'To continue, press RETURN.' ;

END_OF_FORM NAME='HELP_ACCOUNT_DATA' ;
```


Help for Viewing the Account Data
You can view the account data at any time. If you know the secret password, you can change the records--everything but the account number and the date the account was opened.
If you wish to change any account data, enter the secret password and then press RETURN. If you do not give the correct password, you will be returned to the menu.
If the password is accepted, you can change any of the name, address, or phone fields. When you press RETURN (even if you are not in the last field), your changes take effect for the rest of the session. However, they are not made permanently in the SAMP file. (This is, after all, only a simple sample application.) If you press keypad period at any time, no changes are made and you are returned to the menu.
You can return to the menu by pressing RETURN.
For more help, press HELP.
To continue, press RETURN.

```
!*******************************************************!
!       HELP_CHECK                                      !
!       FMS V2 SAMPLE APPLICATION PROGRAM FORM          !
!       Help form for CHECK form                        !
!*******************************************************!

FORM NAME=                      'HELP_CHECK'
    HELP_FORM=                  'HELP_KEYS'
    AREA_TO_CLEAR=              1:23
    WIDTH=                      80
    BACKGROUND=                 CURRENT ;

TEXT (1,2)      'Help for Writing a Check'
                BOLD ;

TEXT (3,2)      'You can write checks up to your current balance, which is '
                &'always at the bottom' ;
TEXT (4,2)      'of the screen.  Just fill in the payee, the amount to pay, '
                &'and, if you like,' ;
TEXT (5,2)      'what the check is for.  Use the TAB and BACKSPACE keys to '
                &'move between fields.' ;

TEXT (7,2)      'The amount will be accepted if it is covered by the balance.'
                &'  The payee and ' ;
TEXT (8,2)      'memo will always be accepted.' ;

TEXT (10,2)     'After you are satisfied with the amount, payee, and memo, '
                &'press RETURN.  The' ;
TEXT (11,2)     'check is recorded in your check register, and the account''s '
                &'balance is updated.' ;

TEXT (13,2)     'Stop the program at any time by pressing keypad period.  Any'
                &' check partially' ;
TEXT (14,2)     'entered will be voided. It will not be entered in your '
                &'register.' ;

TEXT (16,2)     'After writing a check, you will be given the option of '
                &'putting a copy of the' ;
TEXT (17,2)     'check into a check file (SAMPCH.DAT) for later printing.  '
                &'You can then write' ;
TEXT (18,2)     'another check or return to the menu.' ;

DRAW            (20,52) : (23,80);

TEXT (21,53)    'For more help, press HELP.' ;
TEXT (22,53)    'To continue, press RETURN.' ;

END_OF_FORM NAME='HELP_CHECK' ;
```

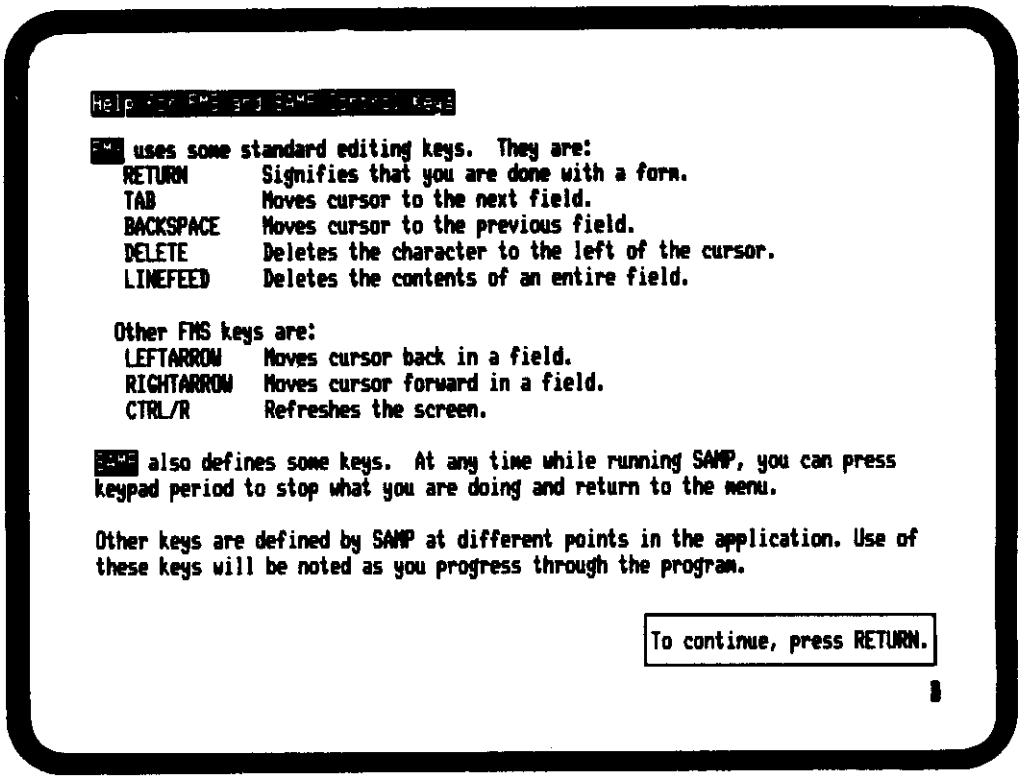

Help for Writing a Check
You can write checks up to your current balance, which is always at the bottom of the screen. Just fill in the payee, the amount to pay, and, if you like, what the check is for. Use the TAB and BACKSPACE keys to move between fields.
The amount will be accepted if it is covered by the balance. The payee and memo will always be accepted.
After you are satisfied with the amount, payee, and memo, press RETURN. The check is recorded in your check register, and the account's balance is updated.
Stop the program at any time by pressing keypad period. Any check partially entered will be voided. It will not be entered in your register.
After writing a check, you will be given the option of putting a copy of the check into a check file (SAMPCH.DAT) for later printing. You can then write another check or return to the menu.
For more help, press HELP.
To continue, press RETURN.

```
!*******************************************************!
!       HELP_DEPOSIT                                    !
!       FMS V2 SAMPLE APPLICATION PROGRAM FORM          !
!       Help Form for DEPOSIT Form                      !
!*******************************************************!

FORM NAME=                      'HELP_DEPOSIT'
    HELP_FORM=                  'HELP_KEYS'
    AREA_TO_CLEAR=              14:23
    WIDTH=                      80
    BACKGROUND=                 CURRENT ;

DRAW (14,1) : (14,80);
TEXT (15,2)     'Help for Making a Deposit'
                BOLD ;
TEXT (16,2)     'To make a deposit, you enter the amount of the deposit and, '
                &'if you wish, a' ;
TEXT (17,2)     'recordof where you got the money.  After entering the '
                &'amount, press the TAB' ;
TEXT (18,2)     'key to go to the memo field.  Press BACKSPACE to go back to '
                &'the amount field.' ;
TEXT (19,2)     'When you are satisfied with both fields, press RETURN.  You '
                &'can press keypad' ;
TEXT (20,2)     'period at any time to return to the menu without completing '
                &'the deposit.' ;

DRAW (21,25) : (23,80);

TEXT (22,26)    'For more help, press HELP.  To continue, press RETURN.' ;

END_OF_FORM NAME='HELP_DEPOSIT' ;
```

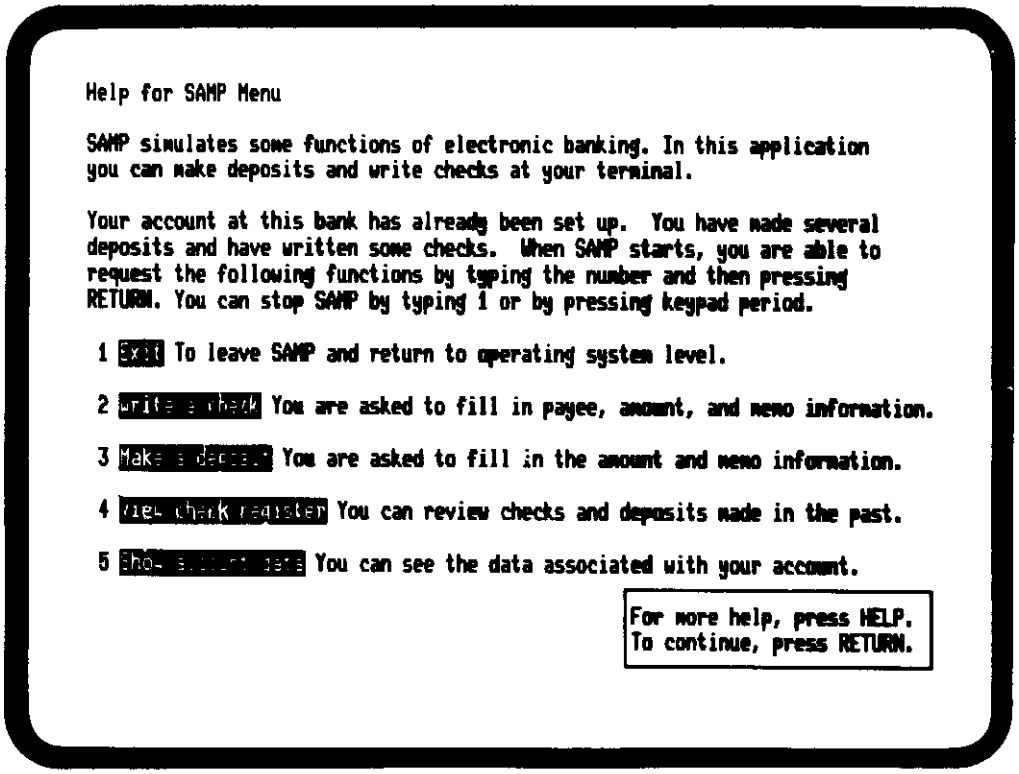


This form shows data supplied by the Sample Application program.

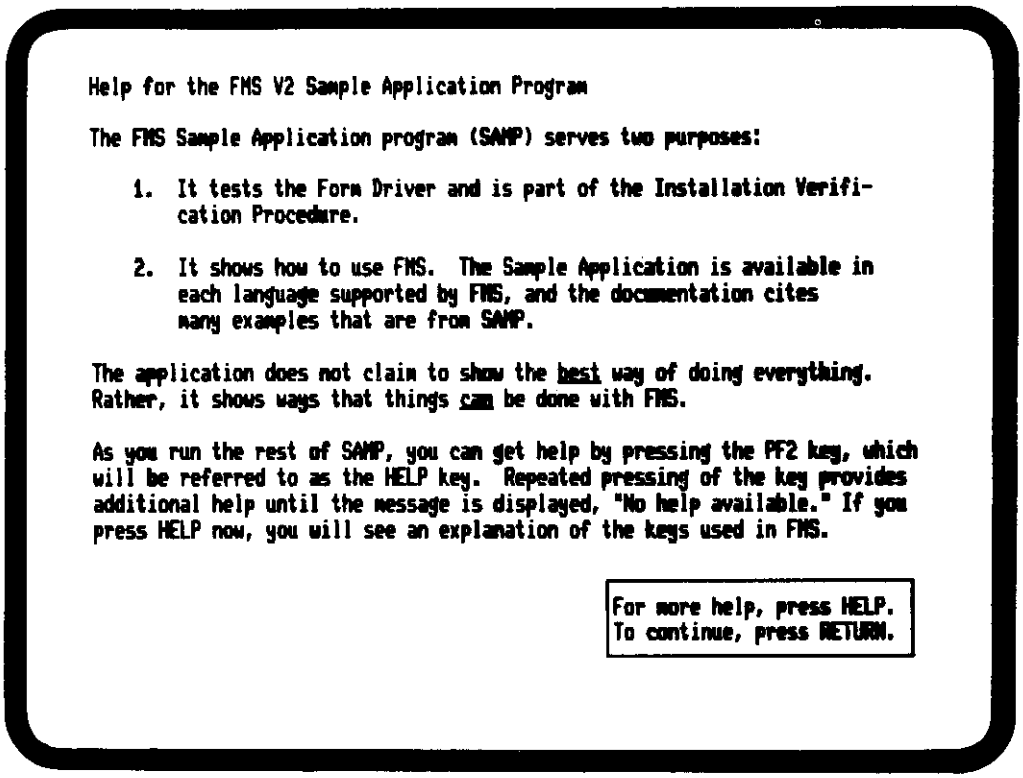
```
!*******************************************************!
!       HELP_KEYS                                       !
!       FMS V2 SAMPLE APPLICATION PROGRAM FORM          !
!       Help for describing FMS keys.                   !
!       Usually last help form in a series.             !
!*******************************************************!

FORM NAME=                          'HELP_KEYS'
    AREA_TO_CLEAR=                  1:23
    WIDTH=                          80
    BACKGROUND=                     CURRENT ;

TEXT (1,2)      'Help for FMS and SAMP Control Keys'
                BOLD ;
TEXT (3,2)      'FMS'
                REVERSE ;
TEXT (3,6)      'uses some standard editing keys.  They are:' ;
TEXT (4,5)      'RETURN       Signifies that you are done with a form.' ;
TEXT (5,5)      'TAB          Moves cursor to the next field.' ;
TEXT (6,5)      'BACKSPACE    Moves cursor to the previous field.' ;
TEXT (7,5)      'DELETE       Deletes the character to the left of the cursor.';
TEXT (8,5)      'LINEFEED     Deletes the contents of an entire field.' ;
TEXT (10,4)     'Other FMS keys are:' ;
TEXT (11,5)     'LEFTARROW    Moves cursor back in a field.' ;
TEXT (12,5)     'RIGHTARROW   Moves cursor forward in a field.' ;
TEXT (13,5)     'CTRL/R       Refreshes the screen.' ;
TEXT (15,2)     'SAMP'
                REVERSE ;
TEXT (15,7)     'also defines some keys.  At any time while running SAMP, you'
                &' can press' ;
TEXT (16,2)     'Keypad period to stop what you are doing and return to the '
                &'menu.' ;
TEXT (18,2)     'Other keys are defined by SAMP at different points in the '
                &'application. Use of' ;
TEXT (19,2)     'these keys will be noted as you progress through the program'
                &'.' ;

DRAW (21,53) : (23,80);

TEXT (22,54)    'To continue, press RETURN.' ;

END_OF_FORM NAME='HELP_KEYS' ;
```


Help for FMS and SAMP Special Keys
FMS uses some standard editing keys. They are:
RETURN Signifies that you are done with a form.
TAB Moves cursor to the next field.
BACKSPACE Moves cursor to the previous field.
DELETE Deletes the character to the left of the cursor.
LINEFEED Deletes the contents of an entire field.
Other FMS keys are:
LEFTARROW Moves cursor back in a field.
RIGHTARROW Moves cursor forward in a field.
CTRL/R Refreshes the screen.
SAMP also defines some keys. At any time while running SAMP, you can press keypad period to stop what you are doing and return to the menu.
Other keys are defined by SAMP at different points in the application. Use of these keys will be noted as you progress through the program.
To continue, press RETURN.

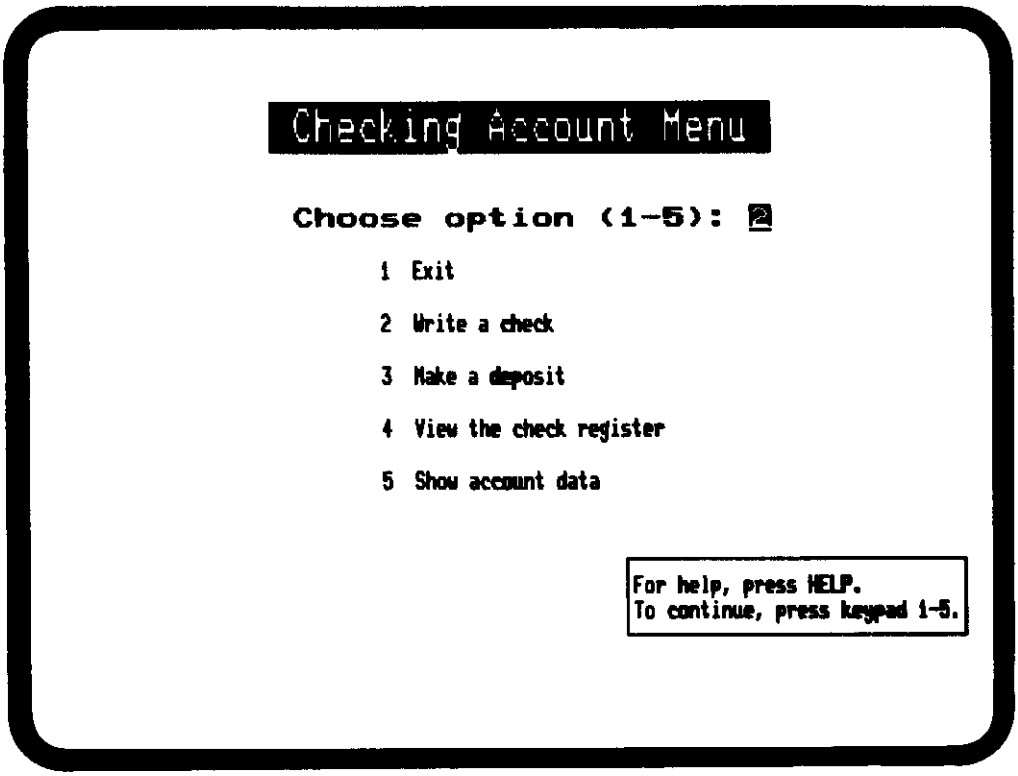
```
!*******************************************************!
!       HELP_MENU                                       !
!       FMS V2 SAMPLE APPLICATION PROGRAM FORM          !
!       Help Form for the MENU Form                     !
!*******************************************************!

FORM NAME=                      'HELP_MENU'
    HELP_FORM=                  'HELP_KEYS'
    AREA_TO_CLEAR=              1:23
    WIDTH=                      80
    BACKGROUND=                 CURRENT ;

TEXT (1,2)      'Help for SAMP Menu'
                BOLD ;
TEXT (3,2)      'SAMP simulates some functions of electronic banking. In this'
                &' application' ;
TEXT (4,2)      'you can make deposits and write checks at your terminal.' ;
TEXT (6,2)      'Your account at this bank has already been set up.  You have'
                &' made several' ;
TEXT (7,2)      'deposits and have written some checks.  When SAMP starts, '
                &'you are able to' ;
TEXT (8,2)      'request the following functions by typing the number and '
                &'then pressing' ;
TEXT (9,2)      'RETURN. You can stop SAMP by typing 1 or by pressing Keypad '
                &'Period.' ;
TEXT (11,3)     '1' ;
TEXT (11,5)     'Exit'
                REVERSE ;
TEXT (11,10)    'To leave SAMP and return to operating system level.' ;
TEXT (13,3)     '2' ;
TEXT (13,5)     'Write a check'
                REVERSE ;
TEXT (13,19)    'You are asked to fill in payee, amount, and memo information'
                &'.' ;
TEXT (15,3)     '3' ;
TEXT (15,5)     'Make a deposit'
                REVERSE ;
TEXT (15,20)    'You are asked to fill in the amount and memo information.' ;
TEXT (17,3)     '4' ;
TEXT (17,5)     'View check register'
                REVERSE ;
TEXT (17,25)    'You can review checks and deposits made in the past.' ;
TEXT (19,3)     '5' ;
TEXT (19,5)     'Show account data'
                REVERSE ;
TEXT (19,23)    'You can see the data associated with your account.' ;

DRAW (20,51) : (23,80) ;

TEXT (21,52)    'For more help, press HELP.' ;
TEXT (22,52)    'To continue, press RETURN.' ;

END_OF_FORM NAME= 'HELP_MENU' ;
```


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.

```
!*******************************************************!
!       HELP_WELCOME                                    !
!       FMS V2 SAMPLE APPLICATION PROGRAM FORM          !
!       Help Form for WELCOME Form                      !
!*******************************************************!

FORM NAME=                      'HELP_WELCOME'
    HELP_FORM=                  'HELP_KEYS'
    AREA_TO_CLEAR=              1:23
    WIDTH=                      80
    BACKGROUND=                 CURRENT ;

TEXT (1,2)      'Help for the FMS V2 Sample Application Program'
                BOLD ;
TEXT (3,2)      'The FMS Sample Application program (SAMP) serves two '
                &'purposes:' ;
TEXT (5,6)      '1.  It tests the Form Driver and is part of the Installation'
                &' Verifi-' ;
TEXT (6,10)     'cation Procedure.' ;
TEXT (8,6)      '2.  It shows how to use FMS.  The Sample Application is '
                &'available in' ;
TEXT (9,10)     'each language supported by FMS, and the documentation cites' ;
TEXT (10,10)    'many examples that are from SAMP.' ;
TEXT (12,2)     'The application does not claim to show the' ;
TEXT (12,45)    'best'
                UNDERLINE  ;
TEXT (12,50)    'way of doing everything.' ;
TEXT (13,2)     'Rather, it shows ways that things' ;
TEXT (13,36)    'can'
                UNDERLINE  ;
TEXT (13,40)    'be done with FMS.' ;
TEXT (15,2)     'As you run the rest of SAMP, you can get help by pressing '
                &'the PF2 key, which' ;
TEXT (16,2)     'will be referred to as the HELP key.  Repeated pressing of '
                &'the key provides' ;
TEXT (17,2)     'additional help until the message is displayed, "No help '
                &'available." If you' ;
TEXT (18,2)     'press HELP now, you will see an explanation of the keys used'
                &' in FMS.' ;

DRAW (20,49) : (23,80) ;

TEXT (21,50)     'For more help, press HELP.' ;
TEXT (22,50)     'To continue, press RETURN.' ;

END_OF_FORM NAME= 'HELP_WELCOME' ;
```

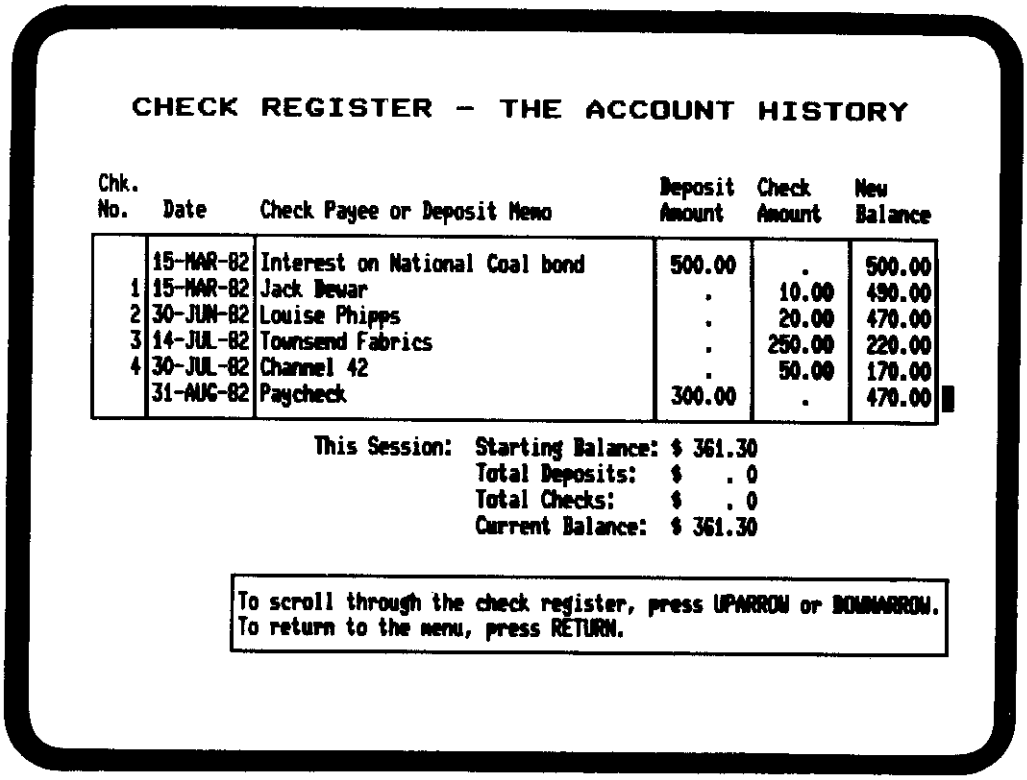

Help for the FMS V2 Sample Application Program
The FMS Sample Application program (SAMP) serves two purposes:
1. It tests the Form Driver and is part of the Installation Verification Procedure.
2. It shows how to use FMS. The Sample Application is available in each language supported by FMS, and the 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.

```
!*********************************************************!
!       MENU                                              !
!       FMS V2 SAMPLE APPLICATION PROGRAM FORM            !
!       Menu                                              !
!*********************************************************!

FORM NAME=                          'MENU'
    HELP_FORM=                      'HELP_MENU'
    AREA_TO_CLEAR=                  1:23
    WIDTH=                          80
    BACKGROUND=                     WHITE
    DBLWID=                         7
    DBLSIZ=                         3
    FUNCTION_KEY_ACTION_ROUTINE='TAKE15' ;


TEXT                    (3,9)    ' Checking Account Menu '
                                 REVERSE ;

TEXT (7,10)             'Choose option (1-5):' ;

FIELD NAME= 'OPTION'    (7,31)   PICTURE= '9'
                                 HELP=
                                   'Enter one of the numbers 1, 2, 3, 4, or 5'
                                 DEFAULT= '2'
                                 ACTION_ROUTINE= 'VALID1' : '12345'
                                 RESPONSE_REQUIRED
                                 UNDERLINE  ;

 TEXT                    (9,27)  '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' ;

 DRAW                    (20,49) : (23,80) ;

 TEXT                    (21,50) 'For help, press HELP.' ;



 TEXT                    (22,50) 'To continue, press Keypad 1-5.' ;

 END_OF_FORM NAME= 'MENU' ;
```


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.

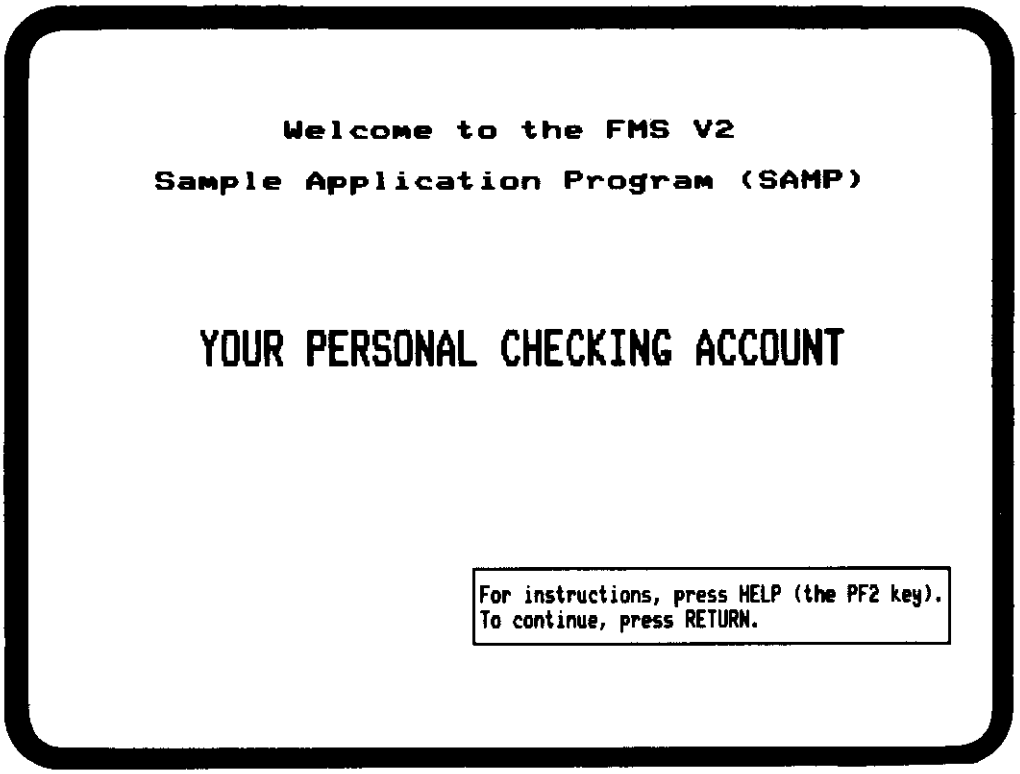
```
!*********************************************************!
!       REGISTER                                          !
!       FMS V2 SAMPLE APPLICATION PROGRAM FORM            !
!       Display the check register and session sums       !
!*********************************************************!

FORM NAME=                          'REGISTER'
    AREA_TO_CLEAR=                  1:23
    WIDTH=                          80
    BACKGROUND=                     CURRENT
    DBLWID=                         2
    FUNCTION_KEY_ACTION_ROUTINE='PASSKY':'110' ; ! Allow only keypad period

! Almost all fields are display only (except field FAKE)

ATTRIBUTE_DEFAULTS FIELD            DISPLAY_ONLY ;

TEXT                        (2,3)   'CHECK REGISTER - THE ACCOUNT HISTORY'
                                    BOLD ;

! Heading for the scrolled area (register proper)

TEXT                        (5,2)
'Chk.                                                       Deposit  Check    New' ;

TEXT                        (6,2)
'No.   Date     Check Payee or Deposit Memo                 Amount   Amount   Balance'

! Note: rather than figure out what the boxes should look like around a
! scrolled area, it makes the most sense to use FED to get the original
! layout of the screen and then edit the back-translated file (which you
! get via FMS/DESC/FULL ...).

TEXT                        (7,1)
'lqqqqwqqqqqqqqqwqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqwqqqqqqqqwqqqqqqqqwqqqqqqqk
                                    CHARACTER_SET= RULE ;

TEXT                        (14,1)
'mqqqqvqqqqqqqqqvqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqvqqqqqqqqvqqqqqqqqvqqqqqqqj
                                    CHARACTER_SET= RULE ;

! Scrolled area definition.
! Note that all text and field definitions start on the first line of the
! scrolled area, line 8.

SCROLL BEGIN_WITH= 8  END_WITH= 13 ;

! These definitions provide the vertical lines in the scrolled area to
! match the lines just above and below the scrolled areas.

TEXT                        (8,1)   'x' CHARACTER_SET= RULE ;
TEXT                        (8,6)   'x' CHARACTER_SET= RULE ;
TEXT                        (8,16)  'x' CHARACTER_SET= RULE ;
TEXT                        (8,53)  'x' CHARACTER_SET= RULE ;
TEXT                        (8,62)  'x' CHARACTER_SET= RULE ;
TEXT                        (8,71)  'x' CHARACTER_SET= RULE ;
TEXT                        (8,79)  'x' CHARACTER_SET= RULE ;

FIELD NAME= 'NUMBER'        (8,2)   PICTURE= 4'X'
                                    RIGHT_JUSTIFIED ;

FIELD NAME= 'DATE'          (8,7)   PICTURE= 'XX-XXX-XX' ;

FIELD NAME= 'PAYMEM'        (8,17)  PICTURE= 35'X' ;
```

```
FIELD NAME= 'DEPOSIT'    (8,54)  PICTURE= 'XXXX.XX'
                                 RIGHT_JUSTIFIED
                                 SUPPRESS ZERO_FILL CLEAR_CHARACTER= '0' ;

FIELD NAME= 'AMTPAY'     (8,63)  PICTURE= 'XXXX.XX'
                                 RIGHT_JUSTIFIED
                                 SUPPRESS ZERO_FILL CLEAR_CHARACTER= '0' ;

FIELD NAME= 'BALANCE'    (8,72)  PICTURE= 'XXXX.XX'
                                 RIGHT_JUSTIFIED
                                 SUPPRESS ZERO_FILL CLEAR_CHARACTER= '0' ;

FIELD NAME= 'FAKE'       (8,80)  PICTURE= 'X'
                                 NODISPLAY_ONLY  !NOTE: ****not display only
                                 NOECHO  ;

! Session information displayed in indexed field.  No good reason, just
! demonstration of indexed fields.

TEXT                     (15,22) 'This Session:  Starting Balance: $' ;
TEXT                     (16,37) 'Total Deposits:    $' ;
TEXT                     (17,37) 'Total Checks:      $' ;
TEXT                     (18,37) 'Current Balance:   $' ;

FIELD NAME= 'SUMMARY'    (15,56) PICTURE= '9999.99'
                                 INDEX= (16,56) : (17,56) : (18,56)
                                 RIGHT_JUSTIFIED
                                 SUPPRESS ZERO_FILL CLEAR_CHARACTER= '0' ;

DRAW                     (20,14) : (23,80);
TEXT                     (21,15) 'To scroll through the check register, press '
                                 &'UPARROW or DOWNARROW.' ;
TEXT                     (22,15) 'To return to the menu, press RETURN.' ;

! Lines in the scrolled area are written with the FDV$PUTSC call so it is
! safest to specify the order of the fields in case the form is ever changed.

ORDER BEGIN_WITH =  1
        NAME= 'NUMBER'
        NAME= 'DATE'
        NAME= 'PAYMEM'
        NAME= 'DEPOSIT'
        NAME= 'AMTPAY'
        NAME= 'BALANCE'
        NAME= 'FAKE' ;

! This Named Data contains the number of scrolled lines in the scrolled
! area so that the program can be more independent of the form.

NAMED_DATA INDEX= 1 NAME= 'NSCROL' DATA= '06' ;

END_OF_FORM NAME= 'REGISTER' ;
```



This form shows data supplied by the Sample Application program.

```
!*********************************************************!
!       WELCOME                                           !
!       FMS V2 SAMPLE APPLICATION PROGRAM FORM            !
!       Initial form: title and welcome message.          !
!*********************************************************!

FORM NAME=                          'WELCOME'
    HELP_FORM=                      'HELP_WELCOME'
    AREA_TO_CLEAR=                  1:23
    WIDTH=                          80
    BACKGROUND=                     BLACK
    DBLWID=                         3:5
    DBLSIZ=                         11
    FUNCTION_KEY_ACTION_ROUTINE='PASSKY' :' ' ; ! No function keys accepted.

TEXT (3,4)       '      Welcome to the FMS V2' ;

TEXT (5,4)       'Sample Application Program (SAMP)' ;

TEXT (11,4)      '  YOUR PERSONAL CHECKING ACCOUNT' ;

! Instruction box

DRAW (20,36) : (23,80) ;

TEXT (21,37) 'For instructions, press HELP (the PF2 key).' ;
TEXT (22,37) 'To continue, press RETURN.' ;

END_OF_FORM NAME= 'WELCOME' ;
```



Appendix C
Sample Application Program Data File

Following is a listing of the SAMP.DAT file that is used in the Sample Application program.

```
53215MAR81Smith                Katherine      M.              1 Hog Hill Rd.          Townsend            AK999998005551212800 5
  15MAR82Interest on National Coal Bond      050000          50000
 115MAR82Jack Dewar                                  1000   49000
 230JUN82Louise Phipps                               2000   47000
 314JUL82Townsend Fabrics                           25000   22000
 430JUL82Channel 42                                  5000   17000
  31AUG82Paycheck                           030000          47000
 512SEP82Four-Star Auto                              1543   45457
 6 4OCT82Mary Johnson                                4450   41007
 7 1FEB83Cory Advertising Agency                     5002   36005
  04FEB83Pegasus Equestrian Center          000125          36130
```

Index

Addressing mode, 3–7
Application program, 1–1, 1–2, 2–1, 3–1, 4–1, 5–1, 6–1, 7–1, 8–1
  compiling, 7–4, 7–6, 7–7, 8–6
  creating, 1–6, 2–9
  creating definition file, 3–9, 4–11, 5–10, 6–10, 7–10, 8–9
  efficiency, 1–5, 2–7, 3–6, 4–9, 5–7, 6–6, 7–7, 8–6
  entry points, 7–7, 8–3, 8–4
  simplifying, 1–3, 2–3, 3–3, 4–3, 5–3, 5–4, 5–8, 6–3, 6–6, 7–3, 7–4, 8–3
Arguments
  null, 1–3, 2–3, 3–3, 4–3, 5–3, 5–4, 6–3, 7–3, 7–4, 8–3
  optional, 1–3, 2–3, 5–3, 6–3, 8–3
  passing, 1–1 to 1–4, 2–1, 2–3, 2–5, 4–5, 5–1, 5–3 to 5–5, 6–1, 6–3, 6–5, 8–1, 8–3, 8–5
    See also Parameters, passing
  required, 1–3, 2–3, 5–3, 5–4, 6–3, 8–3
Arrays, 3–4
  conformant, 7–3
  integer, 6–3, 6–6
  longword integer, 1–4, 2–6, 3–5, 4–7, 4–8, 5–6, 6–5, 7–6, 7–7, 8–6
  one-dimensional, 1–4, 2–5, 2–6, 4–7, 5–6, 6–5, 7–6, 7–7, 8–5, 8–6
  passing, 1–3, 1–4, 2–3, 2–5, 3–3, 3–4, 4–3, 4–4, 5–3, 5–5, 6–3, 6–5, 6–6, 7–3, 7–6, 8–3, 8–5
  word integer, 1–4, 2–6, 3–5, 4–7, 5–6, 6–5, 7–6, 8–6

Call statement. See Procedure call statement
CHARACTER attribute, 8–4
CHARACTER function, 8–9
Character strings, 1–3, 1–4, 1–6, 2–3, 2–6, 2–8, 2–9, 3–3, 3–5, 3–8, 3–9, 4–3, 4–7, 4–10, 5–4, 5–6, 5–8, 5–9, 6–4 to 6–8, 7–5 to 7–9, 8–4, 8–6, 8–8, 8–9
  See also Fixed-length strings; FMS, data types; Varying-length strings
  advantages of, 3–5, 4–7, 6–6
  allocation and deallocation of space, 6–6
  CLASS_S attribute, 7–3
  declaration, 6–6
  declaring length, 11–3
  descriptor, 4–6
  descriptors, 3–3, 3–4, 4–5, 4–6
    advantages of, 4–7
    declaring, 4–6
    initializing, 4–6
  determining length, 1–3, 2–4, 3–3, 3–4, 5–4, 6–4, 7–5, 8–4
  dynamic, 1–3, 2–3, 2–4, 2–6, 3–3, 3–5
  macros, 3–5
  null byte, 4–3
  passing, 1–3, 2–3, 3–3, 4–3, 5–3, 5–4, 6–3, 6–6, 7–3, 8–3
  precautions, 7–6, 7–7, 8–6
  requirements, 1–3, 2–3, 2–4, 2–6, 3–3 to 3–5, 4–3, 4–4, 4–7, 5–4, 6–4, 7–5 to 7–7, 8–4 to 8–6
  string descriptors, 3–3, 3–4, 4–3
CHARACTER VARYING attribute, 8–4
CLASS_S attribute, 7–3
COBOL declarations, 5–6
COBOL library file, creating, 5–6
Command file for building Sample Application, 1–6, 2–2, 2–10, 3–2, 3–10, 4–2, 4–12, 5–2, 5–11, 6–10, 7–10, 8–10

Commands
FMS/DESCRIPTION/BRIEF, 1–3, 2–4, 3–3, 3–4, 5–4, 7–5
FMS/DESCRIPTION/DECLARATIONS, 5–4, 5–6, 6–4, 7–5, 8–4, 8–5
FMS/DIRECTORY/FULL, 1–5, 2–7, 3–6, 4–9, 5–7, 6–6, 7–7, 8–6
COMMON attribute, 2–4, 2–6, 4–8, 6–7
Common storage areas, 1–4, 2–6, 3–5, 3–6, 4–8, 6–6
Compatibility, 6–5
COPY file, 5–2, 5–13
COPY statement, 5–6

Data, displaying, 1–6, 2–8, 2–9, 3–8, 3–9, 4–10, 4–11, 5–8, 5–9, 6–7, 6–8, 7–8, 7–9, 8–8, 8–9
Data conversion, 1–6, 2–8, 2–9, 3–8, 3–9, 4–10, 4–11, 5–8, 5–9, 6–7, 6–8, 7–8, 7–9, 8–8, 8–9
built-in conversion functions, 1–6, 2–8, 2–9, 7–9, 8–8
creating conversion functions, 1–6, 2–8, 2–9, 3–8, 3–9, 4–10, 7–9, 8–8
explicit, 5–8, 5–9
implicit, 5–8, 5–10
macros, 3–8, 3–9
MOVE statement, 5–9
operations, 1–6, 2–8, 2–9, 3–8, 3–9, 4–10, 5–8, 5–9, 6–7, 6–8, 7–9, 8–8, 8–9
PIC 9 variables, 5–10
precautions, 5–9
READV procedure, 7–9
simplifying, 5–8
VAX–11 general conversion routines, 2–9, 3–9, 4–11, 5–9, 6–8, 7–9, 8–9
Debugging aid, 4–6
Debug mode. See FMS, Debug mode
Default, VAX–11, integers, 6–5
Default passing mechanisms, 2–3, 4–3, 5–3, 6–3, 6–6, 8–3
overriding, 5–3
Definition file, 1–6, 2–12, 3–12, 4–14, 5–13, 6–12, 7–12
application program, 2–2
creating, 1–6, 2–9, 3–9, 4–11, 5–10, 5–11, 6–10, 7–10, 8–9
Sample Application, 2–9, 3–9, 4–2, 4–11, 5–2, 5–10, 6–2, 6–8, 6–9, 7–2, 7–9, 7–10, 8–2, 8–9, 8–12
Demonstration program. See Sample Application
%DESC, 6–6
Descriptors, 3–3, 3–4, 4–3 to 4–5
constructing, 4–3 to 4–5
DESCRIP.H (include file), 4–4 to 4–6
use of, 4–4, 4–5

Efficiency, Form Driver, 1–5, 2–7, 3–6, 4–9, 5–7, 6–6, 7–7, 8–6
Entry points, definitions. See Form Driver, entry points
ENVIRONMENT file, 7–4, 7–12
Error message, FDV$_DLN, 2–9, 3–9, 4–11, 5–10, 6–8, 7–9, 8–8
EXTERNAL, 5–8, 8–6 to 8–8
EXTERNAL LONG FUNCTION, 2–2

FDV$ATERM call, 1–5, 2–7, 3–7
FDV$AWKSP call, 1–5, 2–7, 3–7
FDV$READ call, 1–5, 2–7, 3–7
FDV$RETLE call, 1–3, 2–4, 2–5, 3–4, 4–4, 5–4, 5–5, 6–4, 7–5, 8–4, 8–5
FDV$SSRV call, 1–5, 2–7, 3–7
FIXED function, 8–8
Fixed-length strings, 2–3, 2–4, 2–6, 3–3, 3–5, 6–4, 7–5, 7–6, 8–4
See also Character strings, requirements
COMMON attribute, 2–4
MAP statement, 2–4
SPACE$ function, 2–4
FIXED type strings, 3–3
See also Character strings, string descriptors
FMS
language interface. See individual languages
routines. See Form Driver, routines
data types, 1–3, 2–3, 3–3, 4–3, 5–4, 6–4, 7–5, 8–4
See also Character strings
Debug mode, 2–9, 3–9, 4–11, 5–10, 6–8, 7–9, 8–8
status codes, 1–2, 1–5, 2–2, 2–8, 2–10, 3–2, 3–7, 3–10, 4–2, 4–9, 4–11, 5–2, 5–8, 5–11, 6–2, 6–7, 6–10, 7–2, 7–8, 7–10, 8–2, 8–7, 8–10
accessing as functions, 1–2
terminator codes, 2–9, 3–9, 4–11, 5–10, 6–10, 7–10, 8–9
FMS/DESCRIPTION/BRIEF command, 1–3, 2–4, 3–3, 3–4, 5–4, 7–5
FMS/DESCRIPTION/DECLARATIONS command, 5–4, 5–6, 6–4, 7–5, 8–4, 8–5

FMS/DIRECTORY/FULL command, 1–5, 2–7, 3–6, 4–9, 5–7, 6–6, 7–7, 8–6
Form Driver, 1–4, 2–5, 3–4, 4–7, 5–5, 6–5, 7–6, 8–5
  calls, 1–1, 2–1, 2–2, 3–1, 3–2, 4–1, 4–2, 5–1, 5–2, 6–1, 6–2, 7–1, 7–2, 7–5, 8–1, 8–2
    See also Form Driver, routines
    calling sequence, 1–2, 2–2, 3–2, 4–2, 5–2, 6–2, 7–2, 8–2
    data access codes, 1–2, 2–2, 3–2, 4–2, 5–2, 6–2, 7–2, 8–2
    data types, 2–2, 3–2, 4–2, 5–2, 6–2, 7–2, 8–2
    passing mechanisms, 1–2, 2–2, 3–2, 4–2, 5–2, 6–2, 7–2, 8–2
  constants, 7–4
  entry points, 7–4, 7–7, 7–10, 8–3, 8–4, 8–6
  key functions, 2–9, 3–9, 4–11, 5–11, 6–10, 7–10, 8–9
  routines, 1–2, 2–2, 3–2, 3–10, 4–2, 5–2, 6–2, 7–2, 8–2
    See also Form Driver, calls
    codes, 1–6, 2–9, 3–9, 4–11, 5–10, 6–10, 7–10, 8–9
    declarations, 2–10, 3–10, 4–11, 5–11, 6–10, 8–10
  status return methods, 1–2, 2–2, 3–2, 4–2, 5–2, 6–2, 7–2, 8–2
  storage needs, 1–4, 2–6, 3–6, 4–8, 5–7, 6–6, 7–7, 8–6
  using as a shareable image, 3–7
Form Editor, 5–10
Function reference, 2–2, 3–2, 4–2, 6–2, 7–2, 8–2
Functions, 2–10, 3–10, 4–2, 4–11, 5–11, 6–10, 7–10, 8–9, 8–10
  calls, 4–2
  declarations, 2–2, 6–2
  invoking Form Driver routines, 1–2
  key terminators, 2–9, 3–9, 4–11, 5–10, 6–10, 7–10, 8–9
  reference, 1–2

GLOBAL, 3–5, 3–7

%IMMED 0, 7–4
IMPLICIT NONE statement, 6–3
IMPLICIT statement, 6–3
Include file, 4–5 to 4–7, 6–12
  See also Definition file
INHERIT, 7–4
INSPECT statement, 5–9, 5–10
INTEGER, 6–5
INTEGER*2, 6–5
INTEGER*4, 6–2, 6–5
Integer arrays
  %DESCR function, 6–6
  passing, 6–3, 6–6
Integers
  longword binary, 1–3, 2–5, 3–4, 4–4, 5–5, 6–4, 6–5, 7–6, 8–5
  passing, 1–3, 2–3, 3–3, 4–3, 5–3, 6–3, 7–3, 8–3
  word binary, 1–4, 2–5, 3–4, 4–4, 5–5, 6–5, 7–6, 8–5

Key functions. See Form Driver, key functions

Language functions, data conversion, 1–6, 2–8, 2–9, 3–8, 4–11, 7–9, 8–8, 8–9
Language-independent material, 1–1
Language-independent notation, 1–2, 2–2, 3–2, 4–2, 5–2, 6–2, 7–2, 8–2
Language interface. See individual languages
Language issues. See individual languages
Languages. See individual languages
Length of strings. See Character strings, requirements
LOCAL, 3–7

Macros, 3–3, 3–5, 4–5 to 4–8
  data conversion, 3–8, 3–9
  debugging aid, 4–6
  expansion, 4–6, 4–7
MAP statement, 2–4
MLOC allocation. See Run-time memory-resident form area (mloc)
MOVE statement, 5–8, 5–9

Non-FMS data types, 1–4, 2–5, 3–4, 4–7, 5–5, 6–5, 7–6, 8–5
Null arguments, 1–3, 2–3, 3–3, 4–3, 5–3, 5–4, 6–3, 7–3, 7–4, 8–3
  default value, 7–4
  %IMMED 0, 7–4
  syntax, 7–3, 7–4
Numeric arguments, requirements, 1–3, 2–5, 5–5, 6–5, 8–5
Numeric data, 1–6, 2–8, 2–9, 3–8, 3–9, 4–10, 4–11, 5–8, 5–9, 6–7, 6–8, 7–8, 7–9, 8–8, 8–9
Numeric parameters, requirements, 3–4, 4–4, 7–6

Omitted arguments, 1–3, 2–3, 5–3, 5–4, 6–3, 8–3
Omitted parameters, 3–3, 4–3, 7–3, 7–4
Optimizing compiler, 1–5, 2–7, 3–7, 5–8, 6–7, 7–8, 8–7, 8–8
OWN, 3–5 to 3–7

Parameter passing mechanisms. See Parameters, passing
Parameters
  optional, 3–3, 4–3, 7–3, 7–4
  passing, 3–1 to 3–4, 4–1, 4–2, 4–4, 7–1, 7–3, 7–6
    See also Arguments, passing
  required, 3–3, 4–3, 7–3, 7–4
Passing mechanisms. See Arguments, passing; Parameters, passing
PDP–11, compatibility, 6–5
PIC 9 data items, 5–5, 5–8 to 5–10
PIC X, 5–9
PIC X data items, 5–8
Portability, 2–2, 3–2, 4–2, 5–2, 6–2, 7–2, 8–2
Precautions for using FMS
  COMMON attribute, 6–7
  common storage areas, 1–5, 2–7, 5–8
  EXTERNAL attribute, 5–8, 8–7, 8–8
  memory areas, 1–5, 2–7, 3–7, 4–9, 5–7, 6–7, 7–8, 8–7
  OWN or GLOBAL attribute, 3–7
  protection of volatile program variables, 1–5, 2–7, 3–7, 4–9, 4–10, 5–8, 6–7, 7–8, 8–7
  static or external storage areas, 4–9, 4–10
  VOLATILE attribute, 7–7, 7–8
Procedure Calling and Condition Handling Standard. See VAX–11, Procedure Calling and Condition Handling Standard
Procedure call statement, 2–2, 3–2, 5–2, 6–2, 7–2, 8–2, 8–9
  invoking Form Driver routines, 1–2
Procedures. See Procedure call statement
Programming languages, language-specific rules, 1–1, 2–1, 3–1, 4–1, 5–1, 6–1, 7–1, 8–1
  See also individual languages

READ, 6–8
READV procedure, 7–9
REQUIRE file
  See also Definition file
  Sample Application, 3–2, 8–12
Run-time memory-resident form area (mloc), 1–4, 2–6, 3–5, 3–6, 4–8, 5–6, 5–7, 6–6, 7–7, 8–6
  allocation and deallocation of space, 1–5, 2–7, 3–7, 4–9, 4–10, 5–7, 6–7, 7–8, 8–7
  declaration, 1–4, 2–6, 3–6, 4–8, 5–7, 6–6, 7–7, 8–6

Sample Application, 1–6, 2–2, 2–6, 2–9 to 2–11, 3–2, 3–5, 3–9 to 3–11, 4–2, 4–8, 4–11 to 4–13, 5–2, 5–7, 5–10 to 5–12, 6–2, 6–6, 6–8 to 6–11, 7–2, 7–7, 7–9 to 7–11, 8–2, 8–6, 8–9 to 8–11
  command file to compile and link, 1–6, 2–2, 2–10, 3–2, 3–10, 4–2, 4–12, 5–2, 5–11, 6–10, 7–10, 8–10
  COPY file, 5–2, 5–13
  data declarations, 5–6, 5–11
  data file, 1–6
  definition file, 1–6, 2–2, 2–9, 2–12, 3–2, 3–9, 3–12, 4–2, 4–11, 4–14, 5–2, 5–10, 6–2, 6–8, 6–9, 7–2, 7–9, 7–10, 8–2, 8–9
  ENVIRONMENT file, 7–12
  examples, 1–6, 2–2 to 2–9, 3–2 to 3–7, 3–9, 4–2 to 4–4, 4–6, 4–8 to 4–11, 5–1 to 5–10, 6–1 to 6–10, 7–1 to 7–10, 8–1 to 8–2, 8–4 to 8–7, 8–9
  form descriptions and screen images, 1–6
  Include file, 6–12
  REQUIRE file, 8–12
  user action routines, declarators, 5–11
  VAX–11 BASIC, 1–6
SEG$ function, 2–5
Shareable image, Form Driver, 3–7
Simplifying program. See Application program, simplifying
Skeleton declarations, COBOL, 5–6
Standards, VAX–11, 1–1, 2–1, 3–1, 4–1, 5–1, 6–1, 7–1, 8–1
Static storage areas, 5–7
Status codes, 4–2
  accessing returned FMS status codes, 1–2, 2–2, 3–2, 5–2, 6–2, 7–2, 8–2
  FDV$SSRV, 1–5, 2–8, 3–7, 4–9, 5–8, 6–7, 7–8, 8–7
  FDV$STAT, 1–5, 2–8, 3–7, 4–9, 5–8, 6–7, 7–8, 8–7
  FMS, 1–2, 1–5, 2–2, 2–8, 2–10, 3–2, 3–7, 3–10, 4–2, 4–9, 4–11, 5–2, 5–8, 5–11, 6–2, 6–7, 6–10, 7–2, 7–8, 7–10, 8–2, 8–7, 8–10

register zero, 1–2
RMS, 1–5, 2–8, 3–7, 4–9, 5–8, 6–7, 7–8, 8–7
VMS, 2–2, 2–10, 3–2, 3–10, 4–2, 4–11, 5–2, 5–11, 6–2, 6–10, 7–2, 7–10, 8–2, 8–10
Status reporting variables
OWN or GLOBAL storage, 3–7
storage, 1–5, 2–8, 4–9, 5–8, 6–7, 7–8, 8–7, 8–8
Status return methods, 1–2, 2–2, 3–2, 4–2, 5–2, 6–2, 7–2, 8–2
String descriptors. See Character strings, descriptors
SUBSTR function, 8–4, 8–5
VAX–11 PASCAL, 7–5
Syntax, invoking Form Driver routines, 1–2, 2–2, 3–2, 4–2, 5–2, 6–2, 7–2, 8–2
optional arguments, 1–3, 2–3, 5–3, 5–4, 6–3, 8–3
optional parameters, 3–3, 4–3, 7–3, 7–4

TCA allocation. See Terminal control area (tca)
Terminal control area (tca), 1–4, 2–6, 3–5, 3–6, 4–8, 5–6, 5–7, 6–6, 7–7, 8–6
allocation and deallocation of space, 1–5, 2–7, 3–7, 4–9, 4–10, 5–7, 6–7, 7–8, 8–7
declaration, 1–4, 2–6, 3–6, 4–8, 5–7, 6–6, 7–7, 8–6
Terminator codes. See FMS, terminator codes
Truncation, 2–9, 3–9, 4–11, 5–9, 5–10, 6–8, 7–9, 8–8

USAGE IS DISPLAY, 5–4, 5–8
User action routines, 1–3, 2–5, 3–4, 4–4, 5–5, 5–11, 6–4, 7–5, 8–4, 8–5
declarators, 5–11
return codes, 2–10, 3–9, 4–11, 5–11, 6–10, 7–10, 8–9

Varying-length strings, 7–5, 7–6
See also Character strings, requirements
VAX–11
general conversion routines, 2–9, 3–9, 4–11, 5–9, 6–8, 8–9
integers, default, 6–5
Procedure Calling and Condition Handling Standard, 1–2, 2–2, 2–3, 3–2, 4–2, 5–2, 6–2, 6–3, 7–2, 7–3, 8–2, 8–3
standards, 1–1, 2–1, 3–1, 4–1, 5–1, 6–1, 7–1, 8–1
VAX–11 BASIC
FMS language interface requirements, 2–1
FORMAT$ function, 2–8, 2–9
language issues, 2–2
SEG$ function, 2–5
SPACE$ function, 2–4
STR$ function, 2–9
VAL function, 2–8
XLATE function, 2–8, 2–9
VAX–11 BLISS
FMS language interface requirements, 3–1
language issues, 3–2
VAX–11 C
FMS language interface requirements, 4–1
language issues, 4–2
VAX–11 COBOL
FMS language interface requirements, 5–1
language issues, 5–2
VAX–11 FMS interface, 1–1, 2–1, 3–1, 4–1, 5–1, 6–1, 7–1, 8–1
VAX–11 FMS interface, requirements. See individual languages
VAX–11 FORTRAN
FMS language interface requirements, 6–1
language issues, 6–2
READ, 6–8
WRITE, 6–8
VAX–11 languages, syntax requirements. See individual languages
VAX–11 PASCAL
FMS language interface requirements, 7–1
language issues, 7–2
SUBSTR function, 7–5
VAX–11 PL/I
CHARACTER function, 8–9
FIXED function, 8–8
FMS language interface requirements, 8–1
language issues, 8–2
SUBSTR function, 8–4, 8–5
Vectors. See Arrays
VMS status codes, 2–2, 2–10, 3–2, 3–10, 4–2, 4–11, 5–2, 5–11, 6–2, 6–10, 7–2, 7–10, 8–2, 8–10

VOLATILE attribute, 7–7, 7–8
Volatile program variables, 1–5, 2–8, 3–7, 4–9, 4–10, 5–8, 6–7, 7–8, 8–7, 8–8
  FDV$STAT, 1–5, 2–8, 3–7, 4–9, 5–8, 6–7, 7–8, 8–7

WKSP allocation. See Workspace (wksp)
Workspace (wksp), 1–4, 1–5, 2–6, 2–7, 3–5, 3–6, 4–8, 4–9, 5–6, 5–7, 6–6, 7–7, 8–6, 8–7
  allocation and deallocation of space, 1–5, 2–7, 3–7, 4–9, 4–10, 5–7, 6–7, 7–8, 8–7
  declaration, 1–4, 2–6, 3–6, 4–8, 5–7, 6–6, 7–7, 8–6, 8–7
  determining needs, 1–5, 2–7, 3–6, 4–9, 5–7, 6–6, 7–7, 8–6
  estimate of memory needed, 1–4, 1–5, 2–6, 2–7, 3–6, 4–8, 4–9, 5–7, 6–6, 7–7, 8–6, 8–7
WRITE, 6–8

Zero-fill attribute, 5–9, 5–10

HOW TO ORDER ADDITIONAL DOCUMENTATION

From	Call	Write
Chicago	312–640–5612 8:15 AM to 5:00 PM CT	Digital Equipment Corporation Accessories & Supplies Center 1050 East Remington Road Schaumburg, IL 60195
San Francisco	408–734–4915 8:15 AM to 5:00 PM PT	Digital Equipment Corporation Accessories & Supplies Center 632 Caribbean Drive Sunnyvale, CA 94086
Alaska, Hawaii	603–884–6660 8:30 AM to 6:00 PM ET *or* 408–734–4915 8:15 AM to 5:00 PM PT	
New Hampshire	603–884–6660 8:30 AM to 6:00 PM ET	Digital Equipment Corporation Accessories & Supplies Center P.O. Box CS2008 Nashua, NH 03061
Rest of U.S.A., Puerto Rico*	1–800–258–1710 8:30 AM to 6:00 PM ET	
Canada British Columbia	1–800–267–6146 8:00 AM to 5:00 PM ET	Digital Equipment of Canada Ltd 940 Belfast Road Ottawa, Ontario K1G 4C2 Attn: A&SG Business Manager
Ottawa–Hull	613–234–7726 8:00 AM to 5:00 PM ET	
Elsewhere	112–800–267–6146 8:00 AM to 5:00 PM ET	
Elsewhere		Digital Equipment Corporation A&SG Business Manager*

*Prepaid orders from Puerto Rico must be placed with the local DIGITAL subsidiary (call 809–754–7575)

*c/o DIGITAL's local subsidiary or approved distributor

VAX–11 Fl
Language Interfa
Manu
AA–N209A-

READER'S COMMENTS

NOTE: This form is for document comments only. DIGITAL will use comments submitted on this form at the company's discretion. If you require a written reply and are eligible to receive one under Software Performance Report (SPR) service, submit your comments on an SPR form.

Did you find this manual understandable, usable, and well organized? Please make suggestions for improveme

Did you find errors in this manual? If so, specify the error and the page number.

Please indicate the type of user/reader that you most nearly represent.

- ___ Assembly language programmer
- ___ Higher-level language programmer
- ___ Occasional programmer (experienced)
- ___ User with little programming experience
- ___ Student programmer
- ___ Other (please specify) _____

Name _____ Date _____

Organization _____ Telephone _____

Street _____

City _____ State _____ Zip Code _____
or Country

ot Tear — Fold Here and Tape

gital

No Postage
Necessary
if Mailed in the
United States

BUSINESS REPLY MAIL
FIRST CLASS PERMIT NO.33 MAYNARD MASS.

POSTAGE WILL BE PAID BY ADDRESSEE

SSG/ML PUBLICATIONS, MLO5-5/E45
DIGITAL EQUIPMENT CORPORATION
146 MAIN STREET
MAYNARD, MA 01754

Not Tear — Fold Here