VMS Software

VSI OpenVMS

VSI FMS Language Interface Manual

Document Number: DO-FMSLIM-01A

Publication Date: April 2024

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

Software Version: VSI FMS Version 2.6 or higher

VMS Software, Inc. (VSI)
Boston, Massachusetts, USA

VSI FMS Language Interface Manual

Legal Notice

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.

1. About VSI

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

2. 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.

3. Document Structure

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.

4. VSI Encourages Your Comments

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

5. OpenVMS Documentation

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

6. Conventions

The following conventions are used in this manual:

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

Unless specified otherwise, 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 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 the 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 VSI 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 pro gram. 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$GET IDPTIDN$, TERMINAT0RI, '0PTI0N'J
```

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 *VSI 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 L0NG FUNCTI0N F0V$GET

RETURN_STATUS = FDV$GET IDPTI0N$, TERMINATDRZ, 'DPTI0N')
```

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 ( , TERMINATDR%)
```

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$ = '
1194O  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 БPACE$ function pre-extends the string PASS WORD$ to 12 spaces:

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

As an alternative to the above procedures, you may choose to declare fixedlength 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 M AP (ACCOUNT) ACCOUNT$ = 151
220  MAP (ACCOUNT) ACCTNO$ = 5, ACCDATE$ = 7, LAST$ = 20,    &
                   FIRST$ = 15, MIOOLE$ = 15, STREET$ = 30,0 &
                   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 FOV$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 descrip tor.

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 TCAZ (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 WDRKSPACE% (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 memoryresident 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 tor 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 FDM$RET (RI.AMTPAY$.'AMTPAY')
11395  AMTPAY% = OAL(RI.AMTPAY$)
11400  BALANCE% = BALANCE% - AMTPAY%
11405  TOTPAY% = TOTPAY% + AMTPAY%
11410  CALL FDU$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%) + '012345S789')
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 usercreated 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 'BAL ANCE'. 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.

```
$!        SAMPBAS.COM
$!
$!       Compile and link the BASIC version of the FMS V2 Sample Application
$!
$!       The BASIC source files are:      SAMP.BAS
$!
$!       SMP VECTOR.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
```

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 SAMP BLI.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, %ASCID'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. LENGTH FIELD 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 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 $AWKSP routine is called. In the program's declaration section, 12 bytes (3 longwords) are allocated. When FDV$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 memoryresident 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 = .TDTPAY + .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!    SAMPBLI.COM
$!
$!    Compile and link the BLISS version of the FMS V2 Sample Application
$!
$! The BLISS source files are:SAMPBLIBLI
                              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
```

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

In the Sample Application, there are some situations where the $DESCRIPTOR macro is not adequate. Thus, other similar macros are de fined 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 work space 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 memoryresident 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
```

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 SAMP COB.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 TERMINATDR
                     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 "FDM$ 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 "FDVIGET" 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 "FDVIGET" 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 "FOV$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, CHECR_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 runtime 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 memoryresident 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 IN SPECT 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_REGJTEM_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
```

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 fixedlength 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. LENGTH FIELD 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 descrip tor.

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 runtime 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 memoryresident 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 ac count 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 16. 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 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

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
```

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_SJ attribute is used to force use of the [CLASS_SJ 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 non-positional 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 [$11..$ui : 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/DESCRIPTIONIIIECLARATIONS 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.' lt 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 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:

```
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 memoryresident 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 RegisteritemCLastRegisterNumberD 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$EX AMPLES. 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
```

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 SAMP-PLI.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 FOV$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 mes sages 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 array 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 memoryresident 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. AH 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 ac count 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
```

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.dxl,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	*i* 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	*i* 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 OpenVMS 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

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

Appendix C. Sample Application Program Data File

Following is a listing of the SAMP.DAT file that is used in the Sample Application program.