VMS Software

VSI OpenVMS

VSI FMS Form Driver Reference Manual

Document Number: DO-FMSDRM-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 Form Driver Reference Manual

Legal Notice

Preface

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. About This Manual

This manual describes the VSI OpenVMS FMS Form Driver, the run-time component of the Forms Management System for use with the OpenVMS operating system.

3. Intended Audience

The manual is intended for programmers who wish to use FMS with any of their application programs being written in BASIC, BLISS-32, C, COBOL, FORTRAN, PASCAL, or PL/I (those languages documented in the *VSI FMS Language Interface Manual*. Readers are expected to be familiar not only with a programming language but with the OpenVMS system.

Readers are also expected to be familiar with the information of the *VSI FMS Utilities Reference Manual*. This material discusses form characteristics that are specified by means of the Form Editor or Form Language when a form is being designed.

Readers having little or no experience with FMS are urged to read the *Introduction to VSI FMS* before reading this manual.

4. Document Structure

This manual consists of 5 chapter and 1 appendix.

Chapter 1 presents an overview of the Form Driver, briefly discussing 12 basic topics.

Chapter 2 discusses Form Driver interaction with form descriptions, user action routines, terminal operators, key functions and key codes, call status, and asynchronous system traps (ASTs).

Chapter 3 offers programming techniques and examples from the Sample Application Program in various languages. The FMS Sample Application program (SAMP.BAS), a part of the FMS distribution kit, is designed to be a demonstration program and learning tool. (Examples from SAMP appear throughout the FMS document set.)

Chapter 4 shows how to link object modules with the Form Driver and, optionally, with memory-resident forms or user action routines. This chapter also discusses terminal use in FMS programs.

Chapter 5 describes all Form Driver calls in alphabetical order, each giving first the generic format of the call with its arguments, followed by definitions of all arguments in the order they must be specified (along with information on how they are passed and whether they are read or written), then a description of what the call does, and finally a list of status codes (in alphabetical order) that can be returned to the program as a result of the execution of the call.

Throughout this manual, names of calls are usually referred to informally as, for example, CDISP. When calls are actually issued in programs, though, all names of FMS calls begin with the prefix FDV$ (for example, FDV$CDISP).

To find the language-specific way of issuing a call, you must consult the appropriate chapter in the *VSI FMS Language Interface Manual*

In this manual the phrase “GET-type calls” refers to any of the following calls: GET, GETAF, GETAL, and GETSC (but not GETDL). The phrase “PUT-type calls” refers to the calls PUT, PUTAL, PUTD, PUTDA, and PUTSC (but not PUTL).

Appendix A gives a comprehensive summary of information on all FMS calls.

5. 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.

6. OpenVMS Documentation

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

7. Conventions

The following conventions may be used in this manual:

Convention	Meaning
Ctrl/ *x*	A sequence such as **Ctrl/** *x* indicates that you must hold down the key labeled Ctrl while you press another key or a pointing device button.
PF1 *x*	A sequence such as PF1 *x* indicates that you must first press and release the key labeled PF1 and then press and release another key or a pointing device button.
Return	In examples, a key name enclosed in a box indicates that you press a key on the keyboard. (In text, a key name is not enclosed in a box.)
. . .	A horizontal ellipsis in examples indicates one of the following possibilities: • Additional optional arguments in a statement have been omitted. • The preceding item or items can be repeated one or more times. • Additional parameters, values, or other information can be entered.
. . .	A vertical ellipsis indicates the omission of items from a code example or command format; the items are omitted because they are not important to the topic being discussed.
( )	In command format descriptions, parentheses indicate that you must enclose the options in parentheses if you choose more than one.
[ ]	In command format descriptions, brackets indicate optional choices. You can choose one or more items or no items. Do not type the brackets on the command line. However, you must include the brackets in the syntax for OpenVMS directory specifications and for a substring specification in an assignment statement.

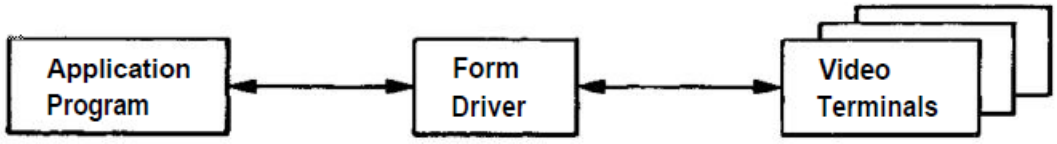
Convention	Meaning
[|]	In command format descriptions, vertical bars separate choices within brackets or braces. Within brackets, the choices are options; within braces, at least one choice is required. Do not type the vertical bars on the command line.
{ }	In command format descriptions, braces indicate required choices; you must choose at least one of the items listed. Do not type the braces on the command line.
bold text	This typeface represents the introduction of a new term. It also represents the name of an argument, an attribute, or a reason.
italic text	Italic text indicates important information, complete titles of manuals, or variables. Variables include information that varies in system output (Internal error *number*), in command lines (/PRODUCER= *name*), and in command parameters in text (where *dd* represents the predefined code for the device type).
UPPERCASE TEXT	Uppercase text indicates a command, the name of a routine, the name of a file, or the abbreviation for a system privilege.
`Monospace type`	Monospace type indicates code examples and interactive screen displays. In the C programming language, monospace type in text identifies the following elements: keywords, the names of independently compiled external functions and files, syntax summaries, and references to variables or identifiers introduced in an example.
-	A hyphen at the end of a command format description, command line, or code line indicates that the command or statement continues on the following line.
numbers	All numbers in text are assumed to be decimal unless otherwise noted. Nondecimal radixes—binary, octal, or hexadecimal—are explicitly indicated.

Chapter 1. Introduction

The Form Driver, the run-time component of FMS, is a subroutine package that is linked with your program. The Form Driver accepts calls from your program, maintains FMS data structures, and issues terminal I/O calls to communicate with the terminal operator.

As shown in Figure 1.1, the Form Driver is one of three parts of an FMS application.

Figure 1.1. Form Driver Communication



Knowledge of the following 12 topics is basic to an understanding of the Form Driver. This chapter offers brief introductions to these topics:

- Terminals, Workspaces, Forms, and Fields
- Terminal Control Areas and Form Workspaces
- Form Management Calls
- Memory-Resident Forms and Form Libraries
- Multiterminal and Multiform Operations
- Debug Mode
- Scrolling Operations
- User Action Routines
- Named Data
- Terminal Key Functions
- Current States
- Operator Aids

1.1. Terminals, Workspaces, Forms, and Fields

The primary work of the Form Driver is the manipulation of terminal images, form workspaces, forms, and form fields.

1.1.1. Terminals

The Form Driver controls one or more terminals, performing such tasks as displaying forms, soliciting data from the terminal operator, and displaying messages. The terminal is controlled by the character sequences and escape sequences sent to it by the Form Driver.

An application program using FMS can process forms on any of the VT100/VT200 compatible terminals or on the VT52 terminal. The Form Driver supports calls that manage terminals' initialization, their use, and their release by FMS.

In an FMS application, a separate area of memory is associated with each terminal. This area is called a terminal control area or TCA. Once a TCA is associated with a terminal, the application program controls that terminal's activity by issuing calls that specify the terminal's TCA.

1.1.2. Workspaces

Workspaces are areas of memory in which the Form Driver stores form descriptions for the forms being processed. More than one form workspace can be associated with a terminal, but each workspace can be associated with only one terminal at a time. The Form Driver supports calls that initialize workspaces, associate them with TCAs, load them with form descriptions, and release them from TCAs.

1.1.3. Forms

A form consists of (1) background text and fields, which are displayed on the screen, and (2) Named Data, which is not displayed. Internally, a form is composed of the data structures used by the Form Driver to create and manipulate the image on the screen. These data structures (called form descriptions) are created by FMS utilities (Form Editor, Form Language Translator, and Form Upgrade Utility). For more information, see the *VSI FMS Utilities Reference Manual*.

The Form Driver supports calls that load these form descriptions into work- spaces and perform the many functions defined for forms and their fields.

When a form is loaded into a workspace, the workspace becomes associated with the form. Consequently, reference to a workspace is a reference to the form stored in it.

Because a workspace can be associated with only one terminal at a time, displaying any given form on more than one terminal requires loading the form into workspaces associated with other terminals as well.

1.1.4. Fields

A field is a portion of the form that has variable data associated with it. The Form Driver supports many calls controlling the manipulation of field data. Some calls allow the terminal operator to enter or change field data. Others, for example, allow the application program to display data in fields, to get data that the operator entered in fields, or to change field video attributes.

Field attributes, assigned when a form is created, affect the way the Form Driver processes field input and output. For example, a field having the Response Required attribute requires that the terminal operator enter at least one character in the field.

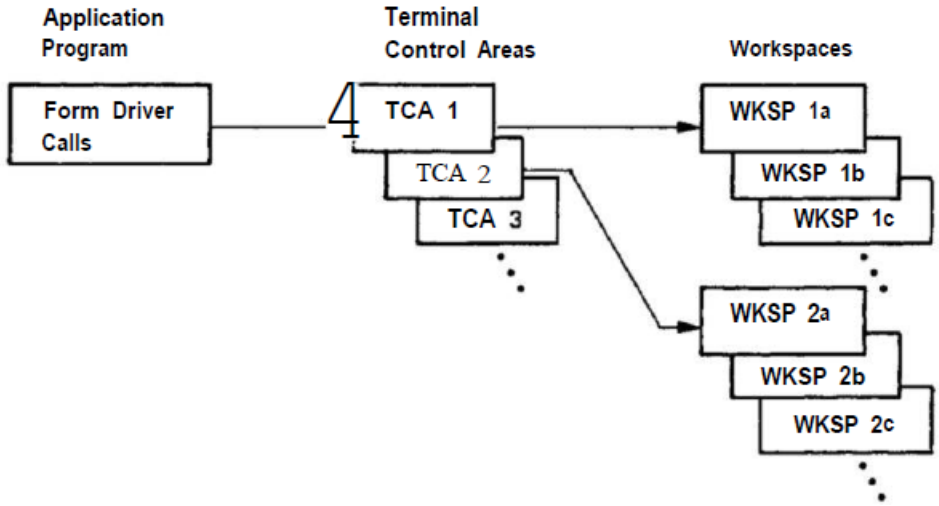
When the Form Driver processes groups of fields (in a single form) with a single call, it processes them in the order specified in the form description. The Form Driver controls movement from field to field, the order in which field values are returned to your program or written to a terminal, and the definition of "first" and "last" fields in forms.

1.2. Terminal Control Areas and Form Workspaces

Terminal control areas (TCAs) and form workspaces are maintained in a hierarchy as shown in Figure 1.2. The application program shown uses several terminals. Attached to the application program

is a list of TCAs that tells the Form Driver which terminals it can use. Attached to the first TCA is a list of workspaces that can contain forms for processing on that TCAs terminal. Similarly, a list of workspaces is attached to the second TCA.

Figure 1.2. Terminal Control Areas and Workspaces



Although you refer to forms and fields by name, you usually do not need to refer to a TCA name to specify a particular terminal or to refer to a workspace name to specify where you want to load a form. The Form Driver establishes a “current terminal” and “current workspace”. Each call that affects a terminal or workspace acts upon the current terminal or current workspace, unless the call explicitly specifies another TCA or workspace.

The Form Driver provides calls to make another terminal or another workspace the current one. By including such calls in your program, you can avoid having to specify the terminal and workspace for each call your program issues.

1.3. Form Management Calls

The Form Driver supports four classes of form management calls:

1. Control
2. Form-level
3. Field-level
4. Utility

These calls are introduced in the following sections and documented fully in Chapter 5.

1.3.1. Control Calls

Your program issues the following calls to make connections and disconnections between FMS units such as terminals, form workspaces, files, and form libraries:

ATERM	Attach terminal
AWKSP	Attach form workspace

DTERM	Detach terminal
DWKSP	Detach form workspace
LCHAN	Set channel for form library file
LCLOS	Close form library
LOPEN	Open form library
STERM	Set current terminal
SWKSP	Set current workspace
TCHAN	Set terminal channel

1.3.2. Form-Level Calls

Your program issues the following calls to perform general form-level operations:

CDISP	Clear screen and display form
DEL	Delete form from memory
DISP	Display form
DISPW	Display loaded form
GETAL	Get all field values
LOAD	Load form without display
NDISP	Mark form in current workspace as not displayed
PUTAL	Output values to all fields in a form
PUTDA	Output default values to all fields in a form
READ	Read form into memory
RETAL	Return values for all fields in a form

1.3.3. Field-Level Calls

Your program issues the following calls to perform field-level operations on a form:

AFCX	Alter field context
AFVA	Alter field video attributes
GET	Get value for specified field
GETAF	Get value for any field
GETSC	Get current line of scrolled area
PFT	Process field terminator
PUT	Output value to specified field
PUTD	Output default to specified field
PUTSC	Output data to current line of scrolled area
RET	Return value for specified field
RETFN	Return current field name
RETFO	Return field names in order
RETLE	Return length of specified field

1.3.4. Utility Calls

Your program issues the following calls to perform operations not falling into the preceding classifications:

ADLVA	Alter data line video attributes
BELL	Ring terminal bell
CANCL	Cancel call
CLEAR	Clear screen
CLEAR_VA	Clear screen video attributes
DFKBD	Define keyboard
DPCOM	Define comma as decimal point
FIX_SCREEN	Repair overwritten lines of terminal screen
GETDL	Get data line from terminal
ILTRM	Return illegal terminators
LEDOF	Turn terminal LED off
LEDON	Turn terminal LED on
PUTL	Output line to screen
RETCX	Return current context
RETDI	Return Named Data by index
RETDN	Return Named Data by name
RETFL	Return form line
RFRSH	Refresh screen
SCR_WIDTH	Set screen width
SIGOP	Signal operator
SPADA	Set keypad to application mode
SPOFF	Turn supervisor-only mode off
SPON	Turn supervisor-only mode on
SSIGQ	Set signal to quiet mode
SSRV	Specify status recording variables
STAT	Return status from last call
STIME	Set field entry timeout
USER_REFRESH	Set up user-supplied refresh routine
WAIT	Wait for operator

1.4. Memory-Resident Forms and Form Libraries

The Form Driver gets the forms required by various calls either from a memory-resident set of form descriptions or from a form library you specify in a Form Driver call. You can make forms memory resident either by building them into your application program or by loading them into the set of

memory-resident forms at run time. Memory-resident form modules are created by one of the Form Application Aids; form libraries are built and maintained by the Form Librarian.

1.5. Multiterminal and Multiform Operations

An application program using the Form Driver can control one or more terminals and manipulate one or more forms on each terminal by using control calls (STERM or SWKSP) to define a new current terminal or current workspace. Note that for this version of FMS you can use these calls to control only one terminal at a time.

1.6. Debug Mode

The Form Driver has a Debug mode of operation that is useful for the debugging of an application program that uses FMS. When the application program is in FMS Debug mode, it runs normally until the Form Driver returns a status code indicating an error. At this time, a message is displayed on the bottom line of the screen, and the Form Driver waits for the operator to press the Enter Form key before proceeding. (Any other input is ignored.)

The Form Driver Debug mode is not associated with any other debugging aid.

1.7. Scrolling Operations

You can use the GET, GETAF, GETSC, PFT, PUT, and PUTSC calls in your program to take advantage of the scrolling capabilities of the VT100. With these calls you can control scrolled areas in forms.

The hardware scrolling capabilities of the VT100/VT200 are simulated for VT52 terminals by software. The performance of scrolling operations is therefore much slower for VT52s.

1.8. User Action Routines

When a form is designed, the form designer can specify that subroutines supplied by you be called from the Form Driver as part of the processing of the form. These subroutines are called user action routines (UARs).

A user action routine can be called under any of the following conditions:

1. When processing for a field is finished
2. When the terminal operator requests help
3. When the terminal operator presses a function key
4. When a screen refresh operation is requested

When linking your program, you must include an object module containing the names of all the user action routines to be called. You generate such an object module by using the FMS/VECTOR command. (See Chapter 4 and the *VSI FMS Utilities Reference Manual*.)

1.9. Named Data

Named Data is data first associated with a form when the form is being designed. It is not a visible part of the form that appears on the operator's screen. Rather, it is form-oriented information your

program can use that you might otherwise have to keep in your program. Two calls are available to you for accessing this information. (See Chapter 3 for examples of the use of Named Data.)

1.10. Terminal Key Functions

Many Form Driver actions are taken in response to the pressing of certain terminal keys by an operator. FMS has assigned functions to keys, but you can change the correspondence between keys and functions to whatever you want. Therefore, this manual refers not to a physical key but to the function that the key performs (regardless of which key implements the function). Accordingly, the name of a key function has its initial letter capitalized rather than its entire name capitalized, which is the way names of actual keys are indicated. (See Section 2.3 for default definitions of all keys used by the Form Driver.)

1.11. Current States

The Form Driver establishes certain “current states” that are important to know about when you are writing your program:

1. Current Terminal — The terminal (with all its characteristics) currently in use

   Although you can control more than one terminal in your FMS application, the Form Driver calls affect only one terminal at a time the current terminal. The Form Driver provides calls to change the current terminal.

   The current terminal is undefined if no terminal is in use.

2. Current Workspace — The form workspace currently in use

   Although you can work on more than one workspace in your FMS application, the Form Driver calls affect only one workspace at a time the current workspace. Your program issues calls to change the current workspace. The current workspace is associated with the current terminal. Whenever your program switches to another terminal, therefore, it inherits the new terminal’s current workspace.

   The current workspace is undefined if the current terminal is undefined, or if no workspace is presently associated with the current terminal.

   You can manipulate a form only by first loading it into a workspace by means of a LOAD, DISP, or CDISP call.

3. Current Field — The most recent field specified in a call for operator input (or the first modifiable field in a form, if no input call has been executed yet) and any index associated with the field

   The current field normally is available by default when you do not supply a name for the field name argument in a subsequent call.

   The current field and its index are associated with the current workspace. If your program switches to another workspace, it inherits a new current field and index.

   If a field does not have the Indexed attribute, its index is defined as zero.

   The current field and its index are undefined if there are no fields in the form or if no workspace is defined.

4. Current Scrolled Line — The line in any scrolled area that the Form Driver is currently working on

This line is initially the first (top) line in a scrolled area, but you can move up or down in the area to a new current scrolled line by issuing a PFT call in your program.

The current scrolled line is associated with any scrolled area being worked on in the current workspace. The current scrolled line is undefined if the form in the workspace has no scrolled areas.

5. Last Terminator — Code The most recent field terminator code returned by a call in your program

   This code is associated with the current workspace. If your program switches to another workspace, it inherits the new workspace's last terminator code.

   The last terminator code is undefined if no field terminator has been entered for the form.

6. Last Status Code — The most recent status code returned by a Form Driver call in your program

   This code is associated with the current workspace if one is defined. If no current workspace is defined, the code is associated with the current terminal provided one is defined. If no current terminal is defined, the code is associated with the Form Driver itself.

   This method of associating the last status code with more than one possible object is available so that when your program switches from one terminal to another, or from one workspace to another, it gets the most recent status code appropriate to its context.

7. Last I/O Status Code — The most recent I/O status code returned by a Form Driver call in your program

   The Form Driver handles this code the same way it handles the last status code.

8. Current Library Channel — The I/O channel to be used any time your program needs access to the associated form library

   For example, you might want to open or close a form library or to retrieve a form for processing. The Form Driver uses the current library channel. The current library channel is associated with the current terminal.

9. Supervisor-Only Flag — The flag used to control the processing of fields having the Supervisor Only attribute

   This flag is associated with the current terminal and is undefined if the current terminal is undefined. The supervisor-only flag is on by default.

10. Timeout Value — The time (in seconds) that an operator has to respond (with each typed character) to a call for input from a terminal

    If the operator fails to respond within the specified time, the call is aborted. This value is associated with the current terminal and is initially zero, meaning no timeout limits are in effect. The timeout value is undefined if the current terminal is undefined.

11. Current Signal Mode — The mode in which the Form Driver gets an operator's attention

    Two modes are available to you ringing the terminal bell and reversing the screen video. The reverse-video signal continues until the operator types a valid character. The current signal mode is associated with the current terminal. The default mode is to ring the bell (the only mode for VT52 terminals).

1.12. Operator Aids

1.12.1. Help

When the Form Driver is waiting for input, the operator can request help by pressing the Help key. Either a help form or a single line of help is available. In either case, the help supplied is specified when the form is designed. Single-line help messages remain on the bottom line of the screen until the operator presses another key.

If a help form is displayed, the Form Driver waits for the operator to press the ENTER key before reconstructing the original screen. If the operator presses the Help key again, instead of ENTER, additional help is displayed if it is available. Any other input is ignored. (The terminal bell rings or the screen video reverses if input has been rejected.)

1.12.2. Screen Refresh

By pressing the Refresh key, the operator can update the screen image. That is, the screen is cleared, and all forms attached to the terminal and displayed are redisplayed. Forms no longer attached to the terminal or marked as not displayed, that were on the screen from a previous display call, are not redisplayed. (You can also issue a RFRSH call from your program.)

A screen refresh also restores the keypad mode, provided your program has previously issued a SPADA call (the Form Driver does not otherwise know the keypad state). The refresh operation also restores the terminal LEDs to the state they were in before the refresh occurred.

Chapter 2. Form Driver Interaction

The Form Driver interacts with your program and with terminals associated with the Form Driver. Discussion of such interactions is concerned with the degree of control your program has in:

- Manipulating forms internally
- Displaying forms on the terminals
- Soliciting terminal operators' responses to requests for field-by-field form data

Some limitation is imposed on both your program and the operator prior to run time, by the way the forms are designed. Thus, for example, field attributes are already assigned for forms, as is the order in which fields are processed in some calls.

Other considerations are concerned with:

- Who has control over the modification of fields in a form, and when
- How control is passed to a terminal operator
- When different levels of help for fields and forms are displayed for an operator
- What keys and keypad layouts can be made available to an operator on VT100-/VT200 and VT52-compatible terminals.

2.1. Interaction with the Form Description

2.1.1. Storing and Accessing Form Descriptions

Your program can store and access form descriptions in either of two ways:

- As disk-resident forms, by reading them directly from a form library file that has been stored on a disk
- As memory-resident forms, for which binary form descriptions (stored as object modules) are linked with the program

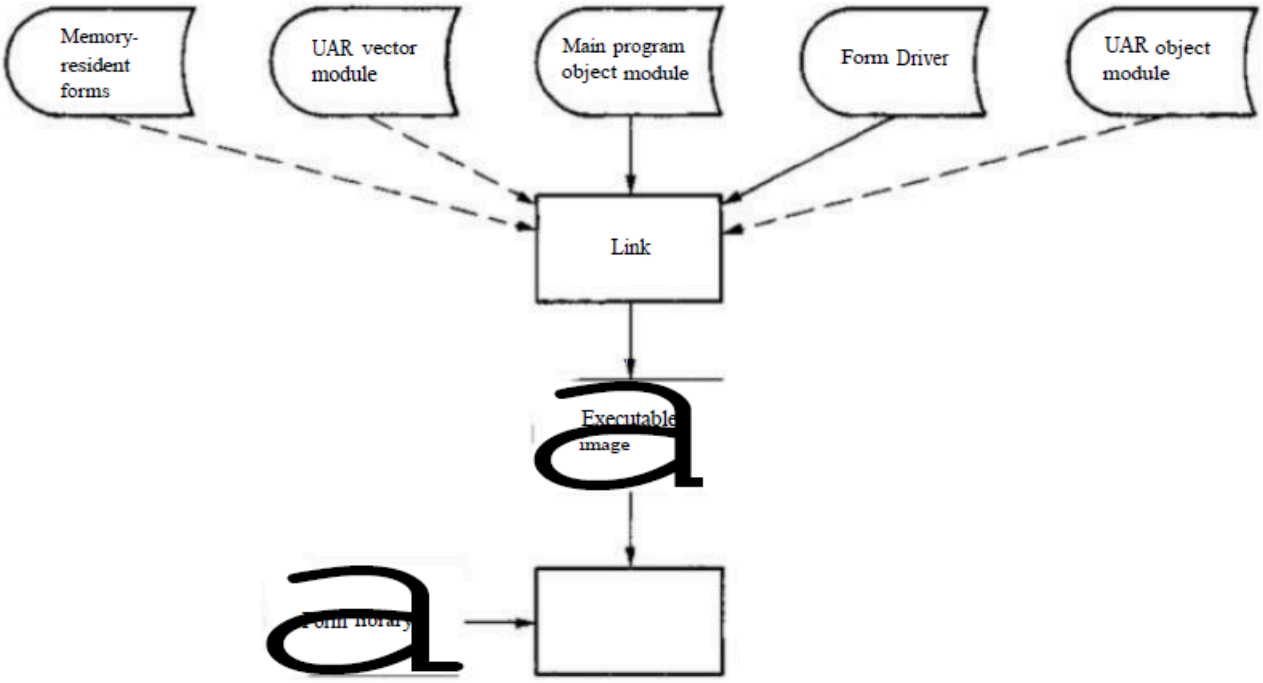
Both ways make use of form descriptions that have been created with the Form Editor, Form Language Translator, Form Upgrade Utility, or Form Converter, and that have been processed with the Form Librarian or Form Application Aids. (See the *VSI FMS Utilities Reference Manual*.)

For example, after using the Form Editor to create a form description, you must use the FMS/LIBRARY/CREATE command to store the description in a form library file or the FMS/MEMORY—RESIDENT command a Form Application Aid — to produce an object module that permits access as a memory-resident form. You must then link the object program with the Form Driver and any other object modules you want to reside in memory, in order to form the executable FMS application program. (See Figure 2.1.)

Note

It is not necessary to build a single large object module containing all memory resident forms for an application. Multiple memory resident form objects produced by FMS can be linked into an executable image.

Figure 2.1. Building an Application Program



2.1.2. Displaying a Form

There are three ways you can access a form. The method you use depends on where the form description resides:

- Resides only in the form library
- Resides in the form library, but is made memory resident at run time
- Is linked with the application program and is memory resident at all times during program execution

A typical procedure for displaying a form at the beginning of an FMS application is:

1. Make a terminal known to the Form Driver. (Use the ATERM call.)
2. Allocate an internal storage area form workspace in which the Form Driver is to store the form description, including field values and other form requirements. (Use the AWKSP call.)
3. If the form is disk resident:
   - Identify the I/O channel the Form Driver is to use for reading form descriptions from the form library file. (Use the LCHAN call.)
   - Open the form library file. (Use the LOPEN call.) [1]
4. If you want that disk-resident form to become memory resident, read the form description into memory. (Use the READ call.)

[1] You can eliminate this step if you include the I/O channel as an input argument in the following LOPEN call.

5. Display the form. (Use the CDISP or DISP call if the form description is disk resident.)

The Form Driver provides two calls that both load workspaces and display forms CDISP and DISP. The CDISP call clears the entire terminal screen before displaying a form. The DISP call clears only the screen lines that are required by the form, in its description, that you want to display.

If you use short forms, you can use the DISP call to create for the operator a screen display that is composed of more than one form or part of a form. In such an instance, only one form would normally be active for the operator, although you could use special techniques like those described in Chapter 3 to switch back and forth among forms.

For each call to display a form, the Form Driver checks the set of memory resident forms first. When memory-resident and disk-resident form descriptions have the same form name, the Form Driver uses only the memory-resident version.

You can load a form from any source into a workspace, but not display the form on the terminal screen, by issuing the LOAD call. You can then put values in fields of the form and display the form later by means of the DISPW call. Use of these calls may reduce the amount of output to the screen, since the fields of the form do not have to be written twice once with the default values and once with the application-defined values.

The name that you assign to a form with the Form Editor or Form Language Translator is the only information that the Form Driver needs to read the form from its form library file or to find its memory-resident description. Similarly, the name that you assign to a field is all the Form Driver requires, regardless of where you locate the field within the form. As long as changes to form and field characteristics have no effect on the logic of your program, you can change the characteristics without having to modify your program.

The following is a description of the screen management role of the Form Driver when overlaid forms are present and when your program issues PUTL and GETDL calls to reference lines that are parts of forms. Whenever the Form—Driver is directed to output a value to the screen by a PUT-type field input fro—m the operator by a GET-type field call — GET, GETAF, GETAL, or the screen. If the form has been disturbed, part or all of it is redisplayed.

A form is disturbed in one of two ways:

1. Part of it has been overlaid by another form in a subsequent DISPW call, RFRSH call or operation, or help request.

2. Part of it has been overlaid by a PUTL or GETDL call.

No matter what part of the form has been overlaid, the form can be redisplayed in its entirety.

The Form Driver functions as a screen manager. It keeps track of every line on the screen belonging to every displayed workspace. Whenever a line is altered through the calls CLEAR, PUTL, GETDL, DISPW, CDISP, or DISP, the Form Driver knows that the line has been affected. If the line is affected by PUTL, GETDL, or CLEAR, the Form Driver knows that the fine has been completely cleared.

If a line is overlaid by the display of another form that clears some lines, then those cleared lines are noted. (A form that overlays another form, without clearing lines, is assumed not to interfere with the underlying form. A form interferes with another form only if it has area-to-clear lines, and then it interferes only with those lines.)

The Form Driver checks the lines of the form when information is sent to a screen field (by a PUT-type call), when information is requested of the screen (by a GET-type call), when help and UAR processing

terminates, or when a new form is displayed. If any line has been cleared, as described above, the Form Driver rewrites all the affected lines of the screen by calling FDV$FIX SCREEN internally. Thus, your program need do nothing if the screen has been affected by calls on the Form Driver, since the Form Driver knows and will fix the screen before the next I/O operation affecting fields.

The Area to Clear attribute of a form is included in the description of the form. Therefore, at design time, the form designer should consider how much of the screen should be included in this attribute. Even if the form does not specify text or fields for a line, the Area to Clear attribute may specify that the line be blank when the form is displayed.

The Form Driver honors the form description whenever the form is referenced. If a form is being redisplayed unexpectedly, it is most likely that part of the form has been overwritten.

2.1.3. Terminal Control

Your application program can use either VT100-/VT200 or VT52-compatible terminals to display forms and to solicit responses from a terminal operator.

Before you can use any terminal to display a form, you must first attach it to the application program by issuing an ATERM call.

More than one terminal can be attached to an application program, but only one terminal at a time can be performing I/O operations. That terminal is called the current terminal. Other attached terminals continue to display any images already on their screens.

At run time, the Form Driver keeps a list of all terminals attached to the application program. Form Driver calls can reference only terminals on this list; a few exceptions use the program's default terminal.

In an FMS application, a terminal control area (TCA) is reserved for each attached terminal. Once a TCA is initialized, the application program controls terminal activity by issuing calls that specify, implicitly or explicitly, the associated terminal control area.

Form Driver calls that manage the initialization of terminals and their release are:

ATERM	Attach a terminal for use by an application program
DTERM	Detach a terminal from the list of attached terminals
STERM	Make the specified terminal the current terminal

You should detach terminals before leaving FMS; otherwise, the terminals may continue to output assigned video attributes when they are no longer associated with FMS. If you do not explicitly detach a terminal from within your program, video attributes assigned to the terminal remain in effect until you reattach the terminal and modify its attributes.

2.1.4. Using Workspaces to Store Forms

To process forms, your program must supply one or more workspaces in memory. Each workspace can contain only one form description. Workspaces are internal to FMS and are inaccessible to the terminal operator.

Each attached terminal can have one or more workspaces associated with it, but the Form Driver can access the form in only one workspace at a time the current workspace. All other attached workspaces are considered passive at that time and can be used only for storing their form descriptions.

Each workspace can be associated with only one terminal at a time. That is, no workspace can be simultaneously associated with more than one terminal. For example, if you want to display a form on two terminals at the same time, you must provide different workspaces for them and load the form into both workspaces.

An advantage of using multiple workspaces is to allow the simultaneous control of multiple forms on the screen. For example, you can display a form stored in one workspace, switch to another form workspace by issuing a SWKSP call and display the form stored there either at the location specified at form definition time or at a screen location appropriately positioned by means of an “offset” argument in the display call. The Form Driver maintains the context of each workspace including, for example, the current field name.

A screen refresh under application control — issuing a RFRSH call — or under operator control — pressing CTRL/R or CTRL/W — clears the screen of all text and redisplays all forms currently marked as displayed for the current terminal.

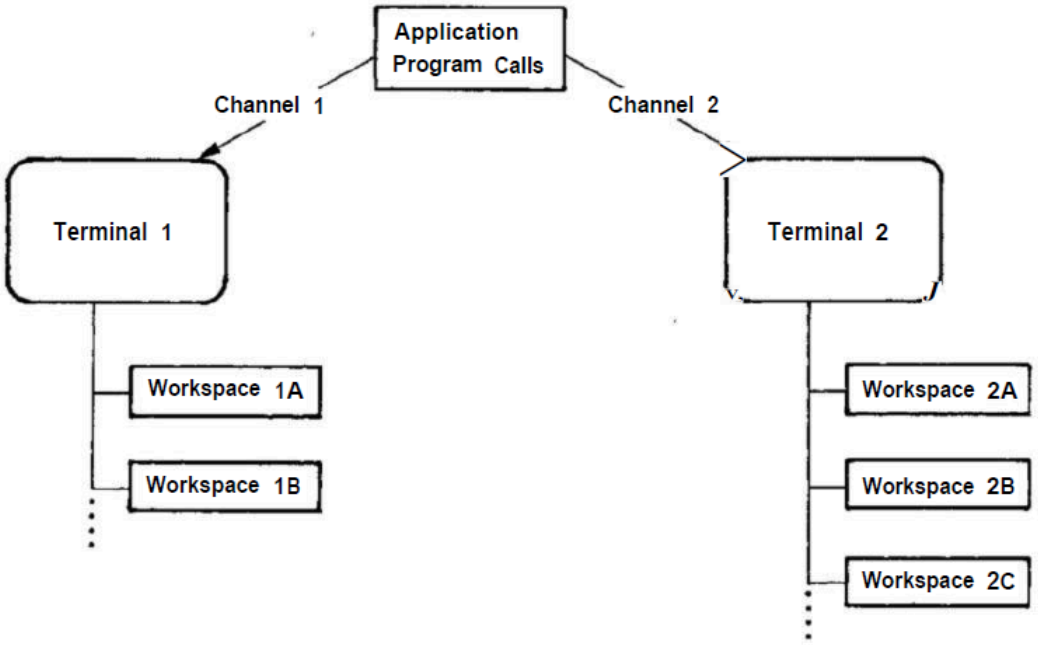
The Form Driver supports calls that associate workspaces with terminals, load workspaces with form descriptions, display forms from workspaces, and release workspaces. These calls are:

AWKSP	Attach a workspace to a terminal control area.
CDISP	Clear the screen and load a workspace with a form from a library or a memory-resident form list; then display the contents of the work space. (The form is marked as being displayed for a later refresh operation.)
DISP	Load a workspace with a form from a library or a memory-resident form list; then display the contents of the workspace. (The form is marked as being displayed for a later refresh operation.)
DISPW	Display the contents of a workspace. (The form is marked as being displayed for a later refresh operation.)
DWKSP	Detach a workspace from the list of attached form workspaces.
LOAD	Read a form description from the library or memory-resident list into the form workspace.
NDISP	Mark a form as being not displayed, but do not delete it from the workspace. (“Not displayed” means not displayed in a refresh operation.)
SWKSP	Make the specified workspace the current workspace.

You can detach workspaces either individually by issuing a DWKSP call or collectively by issuing a DTERM call.

Figure 2.2 shows two VT100-compatible terminals attached to an FMS application. Each terminal must be attached by an ATERM call to the Form Driver in the application program before being used by a terminal operator. Each terminal has attached to it any number of workspaces. Each workspace can store one form description and must be attached to a specific terminal by an AWKSP call before a form description can be stored in it.

Figure 2.2. Attached Terminals and Related Form Workspaces



You maintain control over which form is being updated by specifying the current terminal and the current workspace. The STERM call specifies which terminal is the current terminal. The SWKSP call specifies which of that terminal's workspaces contains the form description to be updated.

To work on a different form in a workspace attached to a terminal that is not the current terminal, you need issue only a SWKSP call, since the Form Driver automatically switches to the associated TCA. For example, if the current workspace is workspace IB and you want to address the form in workspace 2C, you issue the following:

```
CALL FDV45WK5P (WKBP2C)
```

You can determine the current terminal and workspace at any time by issuing a RETCX call to ensure that the proper form is being updated.

2.1.5. The Help Function

Whenever your program issues a call for an operator response, the Form Driver can display two levels of help if the operator requests it help for the field in which the cursor is located and help for the entire form. When the operator presses the Help key once, the Form Driver displays the help text that was specified as a field attribute, if any was specified. Then, when the operator presses the help key again, the Form Driver displays the Help form that was specified as a form-wide attribute, if any was specified.

The operator can erase any help form and have the Form Driver restore the original form at any time. The cursor's position in the original form and all field values are unchanged. If the help form does not overlay the current form, the current form remains on the screen. Otherwise, the help form replaces the portion of the current form that it overlaps.

For each form in your application, both the help text for fields and the help forms have to be specified when the form is created or changed with the Form Editor or the Form Language. Help forms for any disk-resident forms must be stored in the same form library file as the latter.

2.1.6. Field Processing Order

The Form Driver processes multiple fields of a form in the order specified in the form description. This order determines where “next” and “previous” functions take the cursor when the operator presses the corresponding keys to move from field to field. It is also the order in which field values are returned in GETAL, PUTAL, RETAL, GETSC, and PUTSC calls and the derivation of “first” and “last” fields in forms and scrolled lines.

Your program can, nonetheless, control the order in which the operator works with fields. Your program can completely control the access order by issuing a GET call to get the value of a specified field. By repeating this call and specifying different fields, your program requires the operator to complete the fields in the order specified.

Your program can allow the operator partial control by issuing the GETAF call, which allows the operator to choose any field in the form. The operator can respond in only one field, but it can be any modifiable field in the form. Since this call also identifies the name of the completed field, your program can then direct the operator to any other field.

Your program can allow the operator complete control over the order of modifying fields by issuing GETAL, the call for all field values. But the Form Driver returns the field values to your program in a single character string with fields appearing in the order specified in the form description. The Form Driver returns to your application when the operator signals that the entire form is complete.

In the returned field values, the length of each field value is the full length of the field. If the operator enters a value that is shorter than the field, that value is padded out to the field length with the fill character assigned to the field. For a right-justified field, the fill characters precede the value; for a left-justified field, they follow the value.

Two calls, PUTAL and PUTSC, output more than one field value. The PUTAL call specifies new workspace values for all fields in the form and displays the values if the form is displayed. The PUTSC call loads the workspace and displays the field values for one line of a scrolled area.

In both PUTAL and PUTSC, a single character string of field values is written in the order specified in the form description.

2.1.7. Text, Field-Marker Characters, and Video Attributes

After displaying a form, the Form Driver normally is concerned with only the information that relates to the fields, such as the field picture, the fill and clear characters, the default value, and the line of help information. Unless the operator presses the Refresh key, the Form Driver makes no further use of information that is not related to the fields, such as the text in the form, the field-marker characters, or the video attributes of the characters displayed.

In particular, the field values that the Form Driver returns do not contain any of the field-marker characters that the operator sees, such as the hyphen, decimal point, slash, and minus sign. In addition, the field values that your program passes to the Form Driver to display must not include field-marker characters. These field-marker characters are used for display only. They identify certain positions within a field.

2.1.8. Processing Fields

2.1.8.1. Field Pictures

The Form Driver checks the field pictures only when the operator is typing field values. The values that your program passes to the Form Driver for display are not checked for correspondence with the field pictures.

When the operator is responding, a field picture is used to:

- Check that each character satisfies the requirements of the picture character at the corresponding position. For example, in a field that has the mixed picture 999AAA, the Form Driver accepts only digits in the first three positions and only letters in the last three positions.

- Limit the operator's use of the Insert and Overstrike modes of entering field values. For example, the operator cannot change the combination of modes used for a fixed-decimal field or use Insert mode when completing a field that has a mixed picture.

2.1.8.2. Right Justified and Left Justified Field Attributes

The Form Driver uses the Right Justified and Left Justified attributes to:

- Determine the position of the cursor when it is first displayed in a field.

- Align the field value both on the screen and in the form workspace when the value is shorter than its field. The value in a right-justified field always ends at the rightmost character position in a field; the value in a left-justified field always starts at the leftmost character position in a field.

- Determine when the operator has filled a field if the field has the Autotab attribute.

- Set the default mode of entering values in afield. Insert mode is the default for a right-justified field, and Overstrike mode is the default for a left-justified field.

2.1.8.3. Clear Character and Fill Character Attributes

1. The Clear Character and Fill Character attributes affect the way fields whose values do not fill the field are padded on the screen and in the form workspace. The clear character is displayed, and the fill character is inserted as padding in the form workspace. A field with no value is displayed with only the assigned clear character and is stored in the form workspace with only the assigned fill character.

   Where padding an input field value is necessary, the Zero Fill attribute directs the Form Driver to pad with zeros in the form workspace. If Zero Fill is not specified, space characters are used to pad in the workspace.

   The clear character can be any printing character.

2.1.8.4. Default Field Value

When you display a form, the Form Driver displays the default field values and stores them as the current field values in the form workspace. But the Form Editor, the Form Language Translator, and the Form Driver do not check the default values.

Although the Form Editor and the Form Language Translator allow you to assign the numeric default value 13467 for a field with the picture AAAAA, for example, and the Form Driver displays such a value, the Form Driver does not allow the operator to enter the value. Therefore, when developing your application, you must check that the default value is proper for the field.

2.1.8.5. Autotab Attribute

3. When the operator types the character that fills a field having the Autotab attribute, the Form Driver terminates the field as if the operator had pressed the Next Field key. (However, different terminators are returned for fields completed by the Autotab attribute and by the Next Field key.)

   If afield has the Autotab attribute, the Form Driver determines that the field has been filled as follows:

   - For a Must Fill attribute assigned to the field, the operator must have entered enough characters to fill the field.
   - For a left-justified field, the operator must have typed a character in the rightmost character position of the field.
   - For a right-justified field, the leftmost character position must contain a character other than the fill character.

2.1.8.6. Response Required and Must Fill Attributes

The Form Driver checks the validity of an operator's response in a field having the Response Required or Must Fill attribute.

In a field that has the Response Required attribute, the field must contain at least one character other than the assigned fill character before the program will continue to the next field.

In a field that has the Must Fill attribute, the field must contain nothing or must be filled completely. The Form Driver does not accept a field value that is shorter than the field length or a value that contains a fill character.

The occurrence of such checking depends on which call your program issues for an operator response. For the call to get all the form's field values from the operator (GETAL), for example, the Form Driver checks the values for each of these fields when the operator terminates input to the field by pressing the Next Field key or the Enter Form key. In addition, when the operator presses the Enter Form key, the Form Driver checks all modifiable fields in the form.

For other calls, the Form Driver checks the values only when the operator terminates the field with the Next Field key or the Enter Form key.

2.1.8.7. Fixed Decimal Attribute

The Form Driver makes use of the Fixed Decimal attribute to:

- Align the parts of the field value that are to the left and to the right of the decimal point. The Form Driver retrieves input from the part to the left of the decimal point as a right-justified field and the part to the right as a left- justified field.
- Determine the fill and clear characters for the left and right parts of the field. The Form Driver displays the part to the right of the decimal point as if it were in a zero fill, clear character zero, and left-justified field, regardless of whether the Zero Fill or Clear Character attribute is assigned.

The Form Driver applies the assigned Zero Fill or Clear Character attribute only to the part of the value that is to the left of the decimal point. Note that if the fill character for the field is not zero, the Must Fill attribute requires entry and does not allow the usual option of leaving the field empty.

- Determine the position of the cursor when it is first displayed in a field. The cursor is placed at the decimal point, the hanging cursor position for the left part of the field. Output to fixed-decimal fields is treated as if the field were right justified. Such output should not include the decimal point, which is a field marker in fixed-decimal fields.

2.1.8.8. Display Only Attribute

Fields having the Display Only attribute allow you to display their values without letting an operator enter new values. The Form Driver does not allow the operator to position the cursor in a display-only field. When the operator presses the Next Field or the Previous Field key to move the cursor from field to field, the cursor jumps past display only fields as if they were part of the form's background text. Note, however, that data values of these fields are returned from calls such as GETAL and EETAL.

2.1.8.9. No Echo Attribute

The Form Driver uses the No Echo attribute to prohibit field values from being displayed in fields. Not even the clear character or assigned video attributes are displayed in that field. When the operator enters a value in an No Echo field or when your program issues a call to display a field value in an No Echo field, the Form Driver returns the field value to your program and stores it in the form workspace but does not display it.

2.1.8.10. Supervisor Only Attribute

When your program uses the SPON call to turn on the supervisor-only mode, the Form Driver prevents the operator from entering values for fields that have the Supervisor Only attribute. After your program issues the SPON call, the Form Driver treats all fields that have the Supervisor Only attribute as display-only fields. This state, which remains in effect until you issue the SPOFF call, applies to all forms that are displayed on a given terminal. When the program issues the SPOFF call, the Form Driver ignores the Supervisor Only attribute until the program issues the SPON call again.

The initial state for a terminal is SPON. That is, when a terminal is attached, its supervisor-only flag is turned on, thereby disallowing input to fields having the Supervisor Only attribute.

2.1.8.11. Scrolling

Although the Form Editor, the Form Language Translator, and the Form Driver do not allow you to use a form that is longer than 23 screen lines, these components allow you to define sections within a form for displaying portions of large data tables.

A data table is called scrolled because you can "roll" it upward or downward to display the lines that you want the operator to see or to work on. A scrolled area is a window into a form, showing a relatively large amount of data a few lines at a time.

A scrolled area can be as small as one line. Within one form you can define as many separate scrolled areas as will fit within 23 lines. Each line can have as many separate fields as will fit on one screen line. Within each scrolled area, however, all lines must be identical with respect to the number, size, and attributes of fields and all other details.

Because the Form Driver can store field values only for the fields that are on the terminal screen, your program must maintain all scrolled area field values that are not displayed — that is, all the values that

are “above” and “below” each scrolled area. When your program scrolls the lines of a scrolled area upward or downward, the program must collect the lines of values scrolled out of the area and display any line of values scrolled into the area.

Chapter 3 includes some programming examples of scrolled area use.

2.1.8.12. Date and Time Attributes

If a field has one of these attributes, the Form Driver automatically displays the system date or time in the field either when the form is loaded or at any other time when the field default is loaded. The date or time is updated whenever the field default is explicitly loaded; a PUTD or a PUTDA call causes the update, whereas a KFRSH call does not.

Note that only the field default processing is affected; you cannot supply a field default for such a field. You can supply any other field attributes and process the field the way you want, but whenever the field default is displayed by the Form Driver, the date or time is displayed.

2.2. User Action Routines

A user action routine (UAR) is a subroutine that provides special processing during the execution of an FMS application program. A UAR is not in-line code; rather, it is often coded and compiled separately from your program and is called from the Form Driver. A UAR’s object code is later linked with your program’s .OBJ file to form the executable run-time module.

Each UAR is associated with a form or with fields within a form. When a form is designed, the form designer can request that the UAR be called whenever an application program processes that form or field. Then, during execution of the application program, the UAR is automatically called when the terminal operator presses:

- A field terminator key to signal termination of a field or a form (called a field completion UAR)
- The Help key (called a help UAR)
- A function key that is not reserved for FMS (called a function key UAR)

When the UAR ends, it returns a completion code to the Form Driver, indicating whether the routine executed successfully or failed. A UAR can include a return context call (RETCX) to the Form Driver to get any current context information it might require or any other Form Driver call — for example RET — to get values of fields.

The following actions are required before a user action routine can be used:

1. The form designer must use the Form Editor or the Form Language

   Translator to assign a name and associated parameters to the UAR.

2. The form designer must use the appropriate Form Application Aid to create a UAR vector module.
3. The application programmer must code the procedure that the UAR is to perform.
4. The application programmer must link the object module containing the names of all UARs to be called with the application program.

2.2.1. Field Completion UARs

A field completion user action routine is a function routine that is executed at when an Autotab field has been filled or a field terminator key has been pressed.

A field completion UAR is not called under these conditions:

- The field terminator is Previous Field
- The terminator is a function key not reserved for FMS
- The field was terminated with an illegal terminator (see description of ILTRM)

(A function key transformed into a Next Field or an Enter Form terminator by a function key UAR does, however, cause a field completion UAR to be called.)

During the Assign phase of the Form Editor or in using the Form Language Translator, the form designer can identify the name of the UAR and a parameter associated with it. The parameter is a string of up to 80 characters.

Each modifiable field in the form can have up to 15 field completion UARs associated with it. During program execution these UARs are executed each time the operator finishes entering data in a field.

A field completion UAR is also called when your program issues a PFT call with a terminator code of FDV$K FT NTR. This facility lets your program check the validity of each nonscrolled field, just as the Form Driver checks operator data entry for Response Required or Must Fill fields. Note that in a GETAL call, the Enter Form function causes the Form Driver to call all field completion UARs for nonscrolled fields.

All field completion UARs are functions that return status codes. These codes indicate whether or not a UAR performed successfully. The status codes returned by the UAR and the corresponding actions that the Form Driver performs are listed below. The Form Driver interprets any other code as a program_ming error, terminates the GET-type call, and returns a status code of FDV$UAR.

Status codes returned by UARs do not comply with VMS coding standards (for cross-system compatibility reasons).

If a timeout occurs or a CANCL call is issued while the operator is performing afield input operation, the input operation ends immediately, no field completion UARs are called, and any field value returned is undefined.

FDV$K_UVAL_FAIL	Field validation failure. The Form Driver assumes that the field data is in error and requires that the field be reentered. This is the same condition that occurs when the operator types a character that does not agree with the field picture; the Form Driver rejects the input, and the operator is required to reenter the data. No further UARs associated with this field are called if this code is returned.
FDV$K_UVAL_SUC	Field validation success; continue processing. The Form Driver calls any additional action routines associated with this field. If no more exist, the Form Driver completes the field entry normally. If there is nofield entry UAR, the Form Driver acts as if one were called and returns this completion code.
FDV$K_UVAL_END	Field validation success; end further processing of the field. No more UARs associated with this field are called, and the field processing terminates normally.

See Chapter 3 for examples of field completion UARs.

2.2.2. Help UARs

The operator requests help by pressing the key designated as the Help key. There are two times during the processing of help that a UAR can be called. Before the normal Form Driver-supplied help processing begins, the “prehelp” UAR is called. After the normal Form Driver-supplied help processing is exhausted, the “post-help” UAR is called.

2.2.2.1. Pre-Help UAR

A pre-help UAR is called to allow your help text to intercept any help processing provided by FMS. Based on the completion code returned by the UAR, the Form Driver acts as follows:

FDV$K_UHELP_NO	The Form Driver assumes that no help processing was performed and proceeds with the normal help processing. That Help was pressed once already is recorded, and any subsequent pressing of the Help key does not result in the Form Driver’s calling the pre-help UAR again. If no pre-help UAR was specified for the form, the Form Driver acts as if one were called and returns a code of FDV$K_UHELP_NO.
FDV$K__UHELPED	The Form Driver assumes that help was given by the UAR and provides no further help processing for the request. The Form Driver notes that the operator has pressed the Help key at least once for the current field. Any subsequent pressing of the Help key does not, therefore, result in the Form Driver’s calling the pre-help UAR again.
FDV$K_UHELP_ALL	The Form Driver assumes that help was given by the UAR and provides no further help processing for the request. The Form Driver does not record that the operator pressed the Help key once already, and therefore any subsequent pressing of the Help key results in the Form Driver’s calling the pre-help UAR again. This completion code allows a UAR to take over all help processing by ensuring that every time the Help key is pressed, the pre-help UAR is called.

The Form Driver interprets any other code as a programming error. The GET-type call or the WAIT call is terminated, and a status code of FDV$_UAR is returned.

2.2.2.2. Post-Help UAR

This UAR is called to allow your program to supply some form of additional help after the Form Driver-supplied help messages are exhausted. Based on the UAR completion code returned, the Form Driver does the following:

FDV$K_UHELP_NO	The Form Driver issues its HELP EXHAUSTED message. If the operator presses the Help key again, the help sequence starts over again. If no post-help UAR was specified, the Form Driver acts as if it called one and returns a completion code of FDV$K_UHELP_NO.
FDV$K_UHELPED	The Form Driver assumes that the post-help UAR has provided additional help information to the operator. If the operator presses the Help key again, the help sequence starts over again.

FDV$K_UHELP_ALL	The Form Driver assumes that the post-help UAR has provided additional help information to the operator. If the operator presses the Help key again, the UAR is called again.

The Form Driver interprets any other code as a programming error. The call is terminated, and a status code of FDV$_UAR is returned.

The Form Driver allows a depth of 15 in UAR nesting. Help is handled internally as a UAR, so it contributes 1 to the nesting depth each time the HELP key is pressed. A pre or post-help UAR adds another level to the depth.

2.2.3. Help Request Processing

When the Form Driver is processing an input call by your program, the opera tor can request help from the system by pressing the Help key. The current request then stops, and the Form Driver acts as follows:

1. If the operator has not previously pressed the Help key during the processing of the current field, the following actions occur:

    a. The Form Driver calls the pre-help UAR. If the UAR returns the status code FDV $K_UHELP_NO, the Form Driver attempts to display a single help line for the current field. (See the discussion of UARs above for information on UAR status codes.) If no help line exists for the field, the Form Driver goes on to step 2; otherwise, the Form Driver displays the help line, and processing of the Help key is complete. The next time the operator presses the Help key, processing starts with step 2.

    b. If the UAR returns FDV$K_UHELPED as a status code, the Form Driver assumes that the normal single-line help is to be suppressed and that the UAR has provided separate help. As in step la, the next time the operator presses the Help key, the Form Driver begins processing it according to step 2.

    c. If the UAR returns FDV$K_UHELP_ALL, the Form Driver again assumes that no further help is needed, but in addition ends its progression through the help support chain. When the operator presses the Help key again, the Form Driver processes the key according to step 1 again.

2. If step 1 has already been performed, either because a single-line help has already been given or because none could be given and the Form Driver proceeded automatically to this step, the following happens:

    a. If a help form is associated with the current form, it is displayed on the screen. Any subsequent pressing of the Help key begins processing according to step 3.

    b. If no help form is associated with the form displayed, the post-help UAR is called.

    c. If the UAR returns a status code of FDV$K_UHELPED, the Form Driver assumes that the UAR provided help in addition to the forms already displayed. Processing of subsequent pressing of the Help key resumes according to step 1.

    d. If the UAR returns a status code of FDV$K_UHELP_ALL, the Form Driver assumes that the UAR provided help in addition to the forms already displayed. Processing of subsequent pressing of the Help key resumes according to step 2 again.

    e. If the UAR returns a status code of FDV$K_UHELP_NO, the Form Driver assumes that the help available to the operator is exhausted and proceeds according to step 4.

3. If the operator presses the Help key again, the following occurs:

    a. If a help form is associated with the Help form already on the screen, the new help form is displayed. Any subsequent pressing of the Help key begins processing according to step 3 again. (The help form on the screen may have been placed there either during step 2a or by a previous execution of this step.)

    b. If no help form is associated with the help form displayed, the post-help UAR is called.

    c. If the UAR returns a status code of FDV$K_UHELP_ALL, the Form Driver assumes that the UAR provided help in addition to the forms already displayed. Subsequent pressing of the Help key resumes according to step 3 again.

    d. If the UAR returns a status code of FDV$K UHELPED, the Form Driver assumes that the UAR provided help in addition to the forms already displayed. Processing of subsequent pressing of the Help key resumes according to step 1.

    e. If the UAR returns a status code of FDV$K_UHELP_NO, the Form Driver assumes that the help available to the operator is exhausted and proceeds according to step 4.

4. All help is presumed to be exhausted. The Form Driver prints a message indicating that no help is available. Any future Help key processing resumes according to step 1 again, allowing the operator to see the full help sequence again.

Note that when a single help line is displayed, it is displayed on the bottom line of the screen and is removed from the screen the next time a key is pressed. When a help form is displayed, all or part of the screen depending on the help form description is first cleared. The help form is then displayed, and the Form Driver waits for the operator by executing a WAIT call until the operator presses either the Help key or one of the two default Enter Form keys, ENTER or RETURN.

Note

This is one of two occasions in—the Form Driver —the other is the RETURN key as choices for the operator, rather than the more general Enter Form key, which could indicate a redefined alternate key. This arrangement guarantees that you can get out of help mode regardless of how you have redefined keys.

If RETURN or ENTER is returned, the form in the current workspace is returned to its state prior to the displaying of help, and input processing is resumed. If FDV$K_FT_HELP is returned, the operator has pressed the Help key, and the Form Driver behaves according to the description above.

A help UAR can change the screen by putting up new forms in workspaces other than the one current at the time of the UAR call. The Form Driver automatically restores the current workspace and redisplays the current workspace's form if it is overlaid by UAR action.

If a timeout or a cancel condition occurs during help processing, the call is terminated without restoration of any part of the screen. Your program can remove any single-line help from the bottom of the screen by issuing a PUTL call and can restore the rest of the screen by issuing a RFRSH call.

Any call to a PUT or a GET function restores the current workspace's form automatically. Thus, even with a timeout or a cancel condition, your program need not issue a RFRSH call if the same form and workspace are going to be used.

2.2.4. Function Key UARs

A function key UAR can be called whenever the operator presses a key not interpreted by FMS. Function keys include all non data keys that are not FMS functions: control keys, special keys that produce escape sequences — for example, the Uparrow key chapter.

FMS uses some of these keys for editing or for field terminators. You can redefine keys associated with these functions or delete the functions, thus freeing more keys for your program. See Section 2.4.2 for lists of all function keys.

When the operator presses one of these keys during input to a GET-type call or a WAIT call and if the current workspace has a form in it that is marked as being displayed and that has a function key UAR, the Form Driver calls the function key UAR. Further processing depends on the value returned by the UAR, as listed below.

If your program has requested that illegal terminators be returned to it instead of being signaled to the operator (see description of the ILTRM call in Chapter 5), the Form Driver also calls a function key UAR with the illegal terminator. (See Section 2.3.6.8.)

The function key UAR can issue a RETCX call to determine which function key the operator pressed.

Depending on the completion code returned by the UAR, the Form Driver acts as follows:

FDV$K_UKEY_ERR	The Form Driver assumes that no function is defined for the key, the pressing of the key is treated as an illegal keystroke, an error is signaled, and the key is ignored. No field completion UARs are called.
FDV$K_UKEY_TRM	The Form Driver resumes processing of the GET-type call or WAIT call, treating the function key as a key to be returned to your program as a terminator. That is, the code immediately terminates any input operation and is returned as the field terminator code. No field completion UARs are called.
FDV$K_UKEY_NXT	The Form Driver completes field or WAIT processing, treating the function key as if the operator had pressed the Next Field key. If the call is a GET-type call, Must Fill and Response Required attributes are checked, and field completion UARs are called next if any are defined for the field. If this function key results in completion of the entire call, Next Field is recorded as the field terminator character.
FDV$K_UKEY_NTR	The Form Driver completes field or WAIT processing, treating the function key as if the operator had pressed the Enter Form key. If the Form Driver is processing a GET-type call, Must Fill and Response Required field attributes are checked, and field completion UARs are called next if any are defined for the field. This completion code causes Enter Form to be recorded as the field terminator if the operator entered data that was recognized as valid by the Form Driver.
FDV$K_UKEY_SUC	The Form Driver resumes processing of the WAIT or the current field, but otherwise ignores the key, assuming that the UAR successfully performed any processing associated with the key. No field completion UARs are called.

2.2.5. Legal Actions in a UAR

Your program can issue any Form Driver call from a UAR if you observe the following restrictions:

1. You cannot detach the TCA or workspace that is current at the time the UAR is executing.
2. You cannot alter the current workspace by loading a new form into it.

Your program can issue PUT or GET calls, switch to a new workspace, load a new form into a workspace, clear the screen, or take any action you might take if no UAR were to be called. The Form Driver automatically restores the pre-UAR context after the UAR completes its execution. If a UAR changes the screen, the Form Driver ensures that the form in the current workspace is restored to the screen.

The context restored by the Form Driver is:

- Current terminal
- Current workspace
- Current field
- Last terminator entered

Note that the Form Driver does not restore the current field's cursor position or mode of character insertion (Insert or Overstrike). This condition allows a UAR to perform field editing and to reposition the cursor within a field before returning to the Form Driver.

You should be very careful about performing input from a UAR, since this is recursive use of the Form Driver. It is easy to get into an infinite recursion by issuing a GET from a field that calls a field completion UAR that issues a GET, and so on. It is also important to be careful about issuing GET calls from a function key UAR, for the same reason.

The Form Driver allows UARs to be nested to a maximum level of 15. Nesting to a deeper level causes an FDV$_UDP error to be returned to the call that generated the UAR.

2.3. Interaction with the Terminal Operator

The operator has no control until your program allows it by issuing one of the five Form Driver calls for an operator response:

GET	To get the value of a specified field
GETAF	To get the value of a single field that the operator chooses
GETAL	To get all field values for the current form
GETSC	To get all field values for a line in a scrolled area
WAIT	To wait for the operator to enter a terminator

These five calls put the operator in control until the requirements of the call are satisfied. For example, after your program issues the GET call for the value of a specific field, the operator can type and make corrections. The operator also can request help by pressing the Help key. It is only when the operator terminates the field by pressing a field terminator key, such as the Next Field key, that the Form Driver returns control to your program.

Each of the four GET-type calls also returns a status code indicating that a been entered or deleted during the processing of the call. Note, however, that if the operator enters a character and replaces it with the original value, the field is still considered modified.

This section introduces the three kinds of operator activity:

- Correcting errors and requesting help
- Editing fields
- Terminating and choosing fields

2.3.1. Signaling and Recovering from Errors

The Form Driver responds to typing errors and invalid uses of the editing and field termination functions by signaling the operator and displaying messages on the bottom line of the screen.

For all errors, the Form Driver either rings the terminal bell or reverses the video according to the signal mode that is in effect (see description of the SSIGQ call) and ignores the invalid character or function. The Form Driver also displays a one-line explanation at the bottom of the screen. For example, when an operator tries to enter a letter in a field that has been designed to accept only numbers, the Form Driver signals the operator and displays the following message:

```
NUMERIC REQUIRED
```

The *VSI FMS Utilities Reference Manual* lists and explains all messages that can appear.

The Form Driver also provides a Debug mode of operation, which produces a set of error messages that help you in developing and refining your FMS application programs. If you are running your program in Debug mode and an error occurs as the result of a call on the Form Driver, the Form Driver stops your program, signals you, displays the Debug mode message on the bottom of the screen, and waits for you to press the ENTER or the RETURN key, regardless of how you have redefined any keys, before continuing execution of your program.

2.3.1.1. Help Key and Help Messages

The Help function can display two levels of information.

When the operator presses the Help key for the first time, the Form Driver determines whether a help message exists for the current field. If such a one line help message exists, the Form Driver displays it on the last line of the screen. The cursor remains in place within the field.

If the one-line help message is not sufficiently helpful, the operator can press the Help key a second time. The Form Driver then determines whether a help form exists for the current form.

If a help form exists, the Form Driver displays it while saving the context of the current form. Each help form can have yet another help form associated with it that is also displayed.

The operator presses the Enter Form key to return to the original form. In response, the Form Driver restores the form and cursor to what they were before the Help key was pressed.

If no one-line help message for a field exists, the Form Driver displays the help form directly. When no more help is available, the Form Driver displays a message to that effect on the last line of the screen. When the operator next types a field terminator, the Form Driver clears the last line.

Note that although this is the normal sequence for help processing, help UARs can alter it. See Section 2.2.2.

2.3.1.2. Checking Operator Responses from Your Program

Your program must be responsible for checking operator responses at certain times. The Form Driver cannot distinguish valid operator responses from invalid ones in two instances.

First, although the Form Driver accepts only the operator responses that meet the requirements of a field attribute that was assigned with the Form Editor or the Form Language Translator, the Form Driver cannot detect a field value that is invalid in your application. Second, when an operator uses certain function keys to terminate work with a field, the Form Driver does no field checking (Response Required, Must Fill, or Field Completion UARs), leaving that task to your program.

In both of these instances, you can design your program to detect errors and other conditions and to display messages for the operator. Chapter 3 describes some processes and techniques.

2.3.1.3. Refreshing the Screen: Typing CTRL/R

Hold down the CTRL key and press the R key on the keyboard. When the operator types this character, the Form Driver refreshes the screen. That is, the screen is cleared, and all forms currently marked as displayed are redisplayed.

2.3.2. Field Editing Functions

Table 2.1 summarizes the field editing functions that the Form Driver provides and lists the default keys that control the functions. These functions are executed entirely by the Form Driver. You can implement additional functions for the operator by interpreting any function keys not used by FMS. Such functions are implemented by your program after the Form Driver returns control to it.

Table 2.1. Field Editing Keys, Functions, and Usage for the Form Driver

Default Key	Function	Usage
Leftarrow	Cursor Left	Moves the cursor to the preceding data character position within the field, skipping any field-marker character.
Rightarrow	Cursor Right	Moves the cursor to the next data character position within the field, skipping any field marker character.
DELETE	Delete Character	In Insert mode, deletes the character at the cursor's left and closes the space. In Overstrike mode, moves the cursor to the preceding character position within the field, but deletes it only when the character is the last nonblank one in a left justified field.
LINEFEED or F13	Delete Field	Deletes the entire field and resets the mode of character insertion to the default mode for the field (Overstrike or Insert).
PF1 or Blue	Gold Key	Starts a Gold, or 2-character, sequence. Pressing the Gold key several times is equivalent to pressing it once.
PF1 DELETE or Blue DELETE	Reset	Cancels a Gold sequence. If the operator cannot remember if a Gold sequence was started, this sequence safely allows the retyping of the function.

Default Key	Function	Usage
PF1 PF3 or Blue Gray	Insert Mode	Sets Insert mode.
PF3 or Gray	Overstrike Mode	Sets Overstrike mode.
PF, Red or Help	Help	First, displays the help text for the cursor's field and then displays successive help forms for the current form.
CTRL/R	Refresh Screen	Refreshes the screen, with all forms marked as displayed.
Most keyboard keys	Insertion	The keys for the printing characters on the keyboard insert their characters. In the normal, or numeric, keypad mode, the numeric and punctuation keys on the keypad also insert their characters.

2.3.2.1. VT100 Alternate Keypad Mode

You can set the VT100 terminal to an alternate keypad mode or back to a normal (numeric) keypad mode by issuing the SPADA call. Regardless of the terminal's keypad mode, the editing and terminator functions remain the same.

2.3.2.2. The Cursor's initial Position in a Field

The initial position of the cursor in a field depends on whether the field has the Right Justified, Left Justified, or Fixed Decimal field attribute.

For right-justified fields, the initial position is just to the right of the last character position in the field. This position is called the hanging cursor position because the cursor hangs off the end of the field.

For left-justified fields, the initial position is the leftmost character position in the field.

The cursor's initial position for a fixed-decimal field is the decimal point that the Form Driver displays. The decimal point is a field-marker character. It is not stored in the form workspace or returned to your program as part of the field value.

The decimal point in a fixed-decimal field is the rightmost period or comma in the field, whichever one is in effect. Any other periods or commas are treated as normal field-marker characters. A comma is usually used as a decimal point in Europe, and a period is normally used elsewhere.

The Form Driver treats the left part as a right-justified field and the right part as a left-justified field. With the cursor at the initial position, the Form Driver displays the first digits that the operator types in the part to the left of the decimal point until the operator types a decimal point. Then the Form Driver displays the digits that the operator types in the part of the field that is to the right of the decimal point.

As the operator edits a fixed-decimal value, the Delete Field function deletes the entire value and returns the cursor to the initial position. The Delete Character function also deletes the digits in the field value. If the cursor is just to the right of the decimal point, however, the Delete Character function moves the cursor back to the decimal point but does not delete it.

2.3.2.3. Inserting a Field Value: The Default Function

For VT100 keys the Form Driver accepts the standard letters, numbers, and special characters on the keyboard that meet the requirements of the field.

For the keyboard keys, insertion of values infields is the default function. For the numeric and punctuation keys on the keypad, insertion is also the default when the keypad is in the normal, or numeric, mode. In both instances, the operator types values as if using a typewriter.

Insertion is invalid only when it does not meet the field's requirements. For example, letters are invalid where numbers are required. For a field that does not have the Autotab attribute, all characters are invalid when the field is full.

2.3.2.4. The Signed Numeric Picture

A signed numeric picture is treated in two special ways by the Form Driver. One way allows for acceptance of an alternate character for the conventional decimal point. The default is for the Form Driver to allow the period in an N picture and to return it to your pro- gram as part of the value of the field.

For European-style decimal points, your program issues the DPCOM call with an argument of 1. The Form Driver then accepts the comma as the decimal point for the current terminal. After this call, only the comma is recognized as a decimal point in a signed numeric picture. Calling DPCOM with an argument of 0 reestablishes the initial decimal point character, the period. In either case, the decimal point in signed numeric pictures is returned to your program as part of the field value, unlike the decimal point in fixed-decimal fields.

The other special treatment is that the entry of a sign (4 or —) or a decimal point in a field position having an N picture causes the entire field to be checked for valid data. If the field already has a sign or a decimal point, the character is rejected. Thus, any field having an entire signed numeric picture is allowed only one sign and only one decimal point. (A mixed picture field could have more than one sign or decimal point if the additional signs or decimal points were entered into positions that were not signed numeric.)

Note that the Form Driver does not check the position of a sign in a field containing a signed numeric picture; therefore, the sign can occur in the middle rather than at the beginning or the end of the field. You can write afield completion UAR to enforce a particular position for the sign if your application requires it.

2.3.2.5. Deleting a Character

- Default VT100 Key: DELETE

The Delete Character function normally deletes the character that is to the left of the cursor. The function has different effects, however, in Insert and Overstrike modes.

In Insert mode, the Delete Character function deletes the character to the left of the cursor and closes up the space. In a left-justified field, the value remains left justified; in a right-justified field, the value remains right-justified.

In Overstrike mode, the Delete Character function moves the cursor one character to the left. The function does not delete a character in Overstrike mode except when the character is the rightmost character entered in a left justified field.

The Delete Character function is invalid when the cursor is on the leftmost character in a field.

2.3.2.6. Deleting a Field

- Default VT100 Key: LINEFEED
- Default LK201 Key: F13

Regardless of the cursor's position in a field, the Delete Field function deletes all characters in the field except field-marker characters. The Form Driver then displays the assigned clear character for the field and in the form work-space fills the field with the assigned fill character. When the function is comfort a left-justified field, to the right of the rightmost character for a right-justified field, and on the decimal point for a fixed-decimal field. In addition, the mode of character insertion is reset to the default mode for the field (Overstrike or Insert).

The Delete Field function is always valid input in a field.

2.3.2.7. Moving the Cursor to the Right

- Default VT100 Key: -*• (Rightarrow)

The Cursor Right function normally moves the cursor one character to the right within a field. However, the cursor always skips the field-marker characters, such as the hyphen (-) and the slash (/).

The Cursor Right function is invalid when the cursor is to the right of the rightmost character in a field-that is, in the hanging cursor position.

2.3.2.8. Moving the Cursor to the Left

- Default VT100 Key: •*-(Leftarrow)

The Cursor Left function normally moves the cursor one character to the left within a field. However, the cursor always skips the field-marker characters in a field.

The Cursor Left function is invalid when the cursor is on the leftmost character of a field.

2.3.3. Switching the Insertion Modes

- Default VT100 Keys: PF3 on the keypad for Overstrike PF1 and PF3, in sequence, on the keypad for Insert

While theoperator istyping afield value, the Insert and Overstrike insertion modes control how the Form Driver displays the characters. For most of the different types of fields that can be designed, the operator can control the insertion mode by using the Insert and Overstrike functions.

When either the operator or your programfirst moves the cursor to afield, the Form Driver sets the insertion mode according to the attributes of the field. Insert mode is the default for right-justified fields, and Overstrike mode is the default for left-justified fields.

While the operator types in the Insert mode in a left-justified field, the Form Driver inserts each character at the cursor's position. The cursor, the cursor's character, and all characters within the field that are to the right of the cursor are shifted to the right. In a right-justified field, all characters to the left of the cursor are shifted to the left, and the character is inserted to the left of the cursor.

In Overstrike mode, the Form Driver replaces the cursor's character with the character typed and moves the cursor to the right.

In fields that have mixed pictures, Insert mode is invalid. In fixed-decimal fields, the Insert and Overstrike functions are ignored, because of the special data entry conventions that fixed-decimal fields require. In all other instances, the Insert and Overstrike functions are valid.

2.3.4. Field Terminators

Each of the keys listed in Table 2.2 controls a field terminator. The Autotab field attribute also controls a unique terminator. When an operator presses a key that terminates a field or completes afield that

has the Autotab attribute, the Form Driver either processes the terminator itself and displays the effect for the operator or returns a unique field terminator code to your program and leaves the processing to the program. Table 2.2 also gives the processing and code that the Form Driver uses for each field terminator key.

When you set the VT100 *keypad* to alternate keypad mode, the Form Driver also treats the keypad's numeric keys, comma key (,), hyphen key (-), and decimal point key (.) as field terminators. The codes for these alternate keypad mode terminators are returned to your program. In addition, keys not assigned to the Form Driver that are in the following groups are terminators: control keys, all 2-key sequences beginning with the Gold key — PF1 key by default — and all keys producing escape sequences, such as the Uparrow key.

Table 2.2. Default Field Terminator Keys, Values, Symbols, and Effects

Default Key	Value (Decimal)	Symbol	Description
ENTER or RETURN (Enter Form)	0	FDV$K_FT_NTR	Terminates all entries in the form. If the call being processed is a GETAL and if required entries are not complete, the Form Driver refuses to accept the terminator, and the operator remains in control If required entries are complete, the terminator is returned to the program. There fore, the final effect depends on the next call that the program initiates for an operator response. If any other call is being processed, only the requirements for the current field must be satisfied. In such an instance, control is returned to the program.
TAB 1 (Next Field)	1	FDV$K_FT_NXT	Valid only when the current field is not the last modifiable field in the form. Moves the cursor to the initial position of the next field. Processed by the Form Driver for the GETAL and GETSC calls and, until an entry is typed or modified, for the GETAF call. Returned to the program for the GET call and, after an entry is typed or modified, the GETAF call.
	6	FDV$K_FT_SNX	Scroll forward to the next field. The Next Field key or the Autotab attribute in a full field terminated input in the last field of a scrolled line. Always returned to the program.

Default Key	Value (Decimal)	Symbol	Description
BACKSPACE or F12 (Previous Field)	2	FDV$K_FTJPRV	Valid only when the current field is not the first modifiable field in the form. Moves the cursor to the initial position of the previous field. Processed by the Form Driver for the GETAL and GETSC calls and, until an entry is typed or modified, for the GETAF call. Returned to the program for the GET call and, after an entry is typed or modified, the GETAF call.
	7	FDV$K_FTJ3PR	Scroll backward to the previous field. The BACKSPACE key terminated input in the first field in a scrolled line. Always returned to the program.
None (Autotab)	3	FDV$K_FT_ATB	Valid only when the current field is not the last modifiable field in the form. Moves the cursor to the initial position of the next field. Processed by the Form Driver for the GETAL and GETSC calls and, until an entry is typed or modified, for the GETAF call. Returned to the program for the GET call and, after an entry is typed or modified, the GETAF call.
PF1 Uparrow (Exit Scrolled Area Backward)	4	FDV$K_FT_XBK	Valid input only when the current field is in a scrolled area. Moves the cursor out of the scrolled area to the initial position of the previous field that the operator is allowed to complete. Invalid if the scrolled area has the first readable field in the form.
PF1 Downarrow 5 (Exit Scrolled Area Fordward)	5	FDV$K__FT_XFW	Valid input only when the current field is in a scrolled area. Moves the cursor out of the scrolled area to the initial position of the next field that the operator is allowed to complete. Invalid if the scrolled area has the last readable field in the form.
Downarrow (Scroll Forward)	8	FDV$KJVTLSFW	Valid input only when the current field is in a scrolled area. The

Default Key	Value (Decimal)	Symbol	Description
			scrolled area is scrolled up, and the current line remains the same physical line, with new data, or the cursor moves down one line, and that line becomes the new current line. The cursor moves to the initial position of the first field that the operator is allowed to complete in the current line. When processed during a GETAF call, acts like Exit Scrolled Area Forward because GETAF operates only on current scrolled line.
Uparrow (Scroll Backward)	9	FDV$K_FT__SBK	Valid input only when the current field is in a scrolled area. The scrolled area is scrolled down, and the current line remains the same physical line, with new data, or the cursor moves up one line, and that line becomes the new current line. The cursor moves to the initial position of the first field that the operator is allowed to complete in the current line. When processed during a GETAF call, acts like Exit Scrolled Area Backward because GETAF operates only on current scrolled line.

This section describes how your program can use the field terminators and Form Driver calls to guide an operator from field to field in a form in any order.

1. Using the GETAL call

   - The program initiates the GETAL call.
   - The operator uses, at any time, the field terminator keys that move the cursor from field to field nonscrolled fields only. The Form Driver processes these field terminators without returning them to the program.
   - When the operator presses the Enter Form key, the Form Driver checks for valid values for every nonscrolled modifiable field in the form. If a field value is found to be invalid, the Form Driver moves the cursor to the field, and the operator must enter an acceptable value.

     When all values are acceptable, the Form Driver returns the field terminator code and the string of field values to the program. If the operator presses any non-FMS function key, no checking occurs, and the function key code and string of values are returned to the program.
   - The program is then in control of what the operator does next.

2. Using a series of GET calls

- The program initiates the GET call. The operator can type and change only the entry in the specified field. The Form Driver checks for valid field data for any field but one terminated by a function key or the Previous Field key.
- When the operator presses a field terminator key, the Form Driver returns the field terminator code and the single field value to the program. The program then is in control of what the operator does next. For example, on the basis of the field value or the field terminator, the program can specify the same field or another field in the next GET call.

2.3.5. Field Terminators and Form Driver Calls

When your program is sues a call to get an operator response, the Form Driver allows the operator to type an entry in a field or a terminator in response to a WAIT call. When the operator presses a field terminator key that completes the call, the Form Driver passes the field response and the field terminator code to the program and prohibits the operator from further typing. For a WAIT call, the Form Driver accepts any terminator or function key and returns it to your program.

Only the following four Form Driver calls allow the operator to respond in a field:

GET	To get the value and the field terminator for a specified field
GETAF	To get the value for any one field that the operator chooses, as well as the field name and the field terminator
GETAL	To get all field values for the current form and the last field terminator used
GETSC	To get all the field values from the current line of a specified scrolled area and the last field terminator used

For each of these four calls, the Form Driver checks all field terminators. For example, with the cursor in the first field in a form, the Form Driver accepts the field terminator for the Next Field key but does not accept the field terminator for the Previous Field key.

Table 2.3 lists the four calls and shows the field terminator keys that complete each call.

The GET call leaves control of responding to any field terminator to your program.

The GETAF call solicits input for one field but returns control to the program as soon as the operator modifies a field and presses any field terminator key, or presses the Enter Form key or any non-FMS terminator key CTRL key combination, function key, or Gold key sequence. (A non-FMS terminator can always complete a call.)

The GETAL call leaves the Form Driver in control of responding to any field terminator except when the operator presses the Enter Form key.

The GETSC call leaves the Form Driver in control within a line of a scrolled area until the operator presses the Enter Form key or any function key that results in an exit from the scrolled line.

Table 2.3. GET-type Calls and Their Field Terminators

Call	**Field Terminator Keys that Complete the Call**
GET	Any valid field terminator key, the Autotab code, or any non-FMS key

Call	Field Terminator Keys that Complete the Call
GETAF	Enter Form key or any typed field entry followed by any valid field terminator key, the Autotab code, or any non-FMS key
GETAL	Enter Form key or any non-FMS key
GETSC	Enter Form, Scroll Backward, Scroll Forward, Exit Scrolled Area Backward, Exit Scrolled Area Forward, Next Field (or Autotab) out of last field, Previous Field out of first field, or any non-FMS key

The following principles summarize Table 2.1 and Table 2.3:

1. The effects of the field terminator keys cannot be changed from what VSI has designed in the following calls:
   - GETAL
   - GETSC
   - GETAF before the operator makes a field entry
2. When the operator presses the default Enter Form key or, in response to the GET call, any field terminator key, the program alone controls the effect that the operator sees.

For example, if you use the GETAL call in a program, the Next Field key advances the cursor from field to field according to the order that is built into the form description. But if you use a series of GET calls instead of the GETAL call, the program is passed the field terminator code for the Next Field key and can react to it in any way you specify.

Your program can, for example, issue the PFT call. After the operator uses any field terminator that returns control to the application program, the program can initiate the PFT call, making the Form Driver follow the effects of any field terminator key. In the example of a GET call terminated by the operator's pressing of the Next Field key, the program can react by specifying the Previous Field key in the PFT call. Then, the effect of the next GET call is to move the cursor back to the previous field in the form.

Alternatively, your program can issue another GET call. In the example of a GET call terminated by a Next Field function key, the program can react with another GET call that specifies by name the next field that the operator is to complete, regardless of where the field appears on the operator's screen.

2.3.6. Field Terminating Functions

The operator presses terminator keys to move to new fields or a new form. How the Form Driver processes these functions depends on the current Form Driver call that is being executed. In many instances, the Form Driver gives your program an opportunity to intercept and change the terminator function that the operator has used. The Form Driver identifies each terminator function by means of a unique terminator code.

Because the Form Driver can be executed from either a VT52-, a VT100-, or a VT200-compatible terminal, a set of terms common to both devices is required to describe logical field terminating functions. In addition, since your program can modify the association between keys and functions, the field terminators are referred to by their functions rather than by the names of particular keys.

Table 2.1 describes the relationship between the logical function keys referred to in this manual and their corresponding default physical keys on VT52-, VT100-, and VT200-compatible terminals, with an LK201 keyboard.

2.3.6.1. Signaling that the Form Is Complete

- Default VT100/VT200 Keys:

  ENTER on the keypad

  RETURN on the keyboard

- Terminator Code

  Value: 0

  Symbol: FDV$K_FT_NTR

The Enter Form key signals that the operator has completed the current form.

When a GETAL call is issued, the Form Driver does not accept the Enter Form key until all field values satisfy their field requirements. A Response Required field must have a response of at least one character, a Must Fill field must be either empty or filled, and all field completion UARs must return success values.

For any other Form Driver call, control is returned to the program if the requirements for the current field value are satisfied.

2.3.6.2. Moving the Cursor to the Next Field

- Default VT100/VT200 Key: TAB

- Terminator Code

  Value: 1 (when terminating a field outside of a scrolled area)

  Symbol: FDV$K_FT_NXT

  Value: 6 (when terminating a field at the end of a scrolled line)

  Symbol: FDV$K_FT_SNX

The Next Field function is valid only when the requirements for the current field value Response Required, Must Fill, or field completion UARs are satisfied.

The effects of the Next Field function depend on the Form Driver call that is being executed.

For the GETAL and GETSC calls and for the GETAF call before the operator enters or changes a field value, the Form Driver processes the function directly and moves the cursor to the initial position of the next modifiable field.

For GETSC at the end of a scrolled line, control is returned to the program. For the GET call and for the GETAF call after the operator enters or changes a field value, the Form Driver transfers control to the program. The next call in your program determines what the operator sees. For example, after the operator terminates a field with the Next Field key, your program might display a new form, calculate and display a value in a display-only field, or issue another call for another operator response in a specific field.

The function is invalid when the cursor is in the last nonscrolled modifiable field of the form.

2.3.6.3. Moving the Cursor to the Previous Field

- Default VT100 Key: BACKSPACE
- Default LK201 Key: F12
- Terminator Code

  Value: 3 (when terminating a field outside of a scrolled area)

  Symbol: FDV$K_FT_PRV

  Value: 7 (when terminating a field at the beginning of a scrolled line)

  Symbol: FDV$K_FT_SPR

The effects of the Previous Field key depend on the Form Driver call that is being executed.

For the GETAL and GETSC calls and for the GETAF call before the operator enters or changes a field value, the Form Driver processes the function directly and moves the cursor to the initial position of the previous modifiable field.

For GETSC at the beginning of a scrolled line, control is returned to the program. For the GET call and for the GETAF call after the operator enters or changes a field value, the Form Driver transfers control to the program. The next call in your program determines what the operator sees.

The function is invalid when the cursor is in the first nonscrolled modifiable field of the form.

2.3.6.4. Scrolling Backward

- Default VT100/VT200 Key: f (Uparrow)
- Terminator Code

  Value: 9

  Symbol: FDV$K_FT_SBK

The Scroll Backward key is valid only when the cursor is in a field that is within a scrolled area. For GETAF before the operator enters or changes a field value, the Form Driver processes the key directly and moves the cursor to the initial position of the first modifiable field before the scrolled area, as if the key were the Exit Scrolled Area Backward key.

The function transfers control to your program for GET and GETSC and for GETAF after the operator enters or changes a field value. Therefore, you can choose to use the function in any way you want, and the effects that the operator sees depend on the next calls that your program issues.

The Form Driver processes the Scroll Backward terminator when you specify its code in the PFT call. The Form Driver either moves the cursor to the preceding data line within the scrolled area and places the cursor at the initial position of the first modifiable field in that data line or scrolls the area backward and places the cursor at the initial position of the first modifiable field in the current line.

When the cursor is on the top screen line of the scrolled area, or if the program specifies data to update the top line, the Scroll Backward function scrolls the bottom scrolled line of information off the screen,

scrolls a new line of information into the top line of the scrolled area, and moves the intermediate scrolled lines downward. If the cursor is on the top line and if your program specifies values for the new line of information, they are displayed; otherwise, the default field values are displayed.

The function is invalid when the cursor is in a field that is not within a scrolled area.

2.3.6.5. Scrolling Forward

- Default VT100/VT200 Key: | (Downarrow)
- Terminator Code

  Value: 8

  Symbol: FDV$K_FT_SFW

The Scroll Forward function is valid only when the cursor is in a field that is within a scrolled area. For GETAF before the operator enters or changes a field value, the Form Driver processes the key directly and moves the cursor to the initial position of the first modifiable field after the scrolled area, as if the key were the Exit Scrolled Area Forward key.

The function transfers control to your program for GET and GETSC and for GETAF after the operator enters or changes a field value. Therefore, you can choose to use the function in any way you want, and the effects that the operator sees depend on the next calls that your program issues.

The Form Driver processes the Scroll Forward key when you specify its code in the PFT call. The Form Driver either moves the cursor to the next data line within the scrolled area and places the cursor at the initial position of the first modifiable field in that data line or scrolls the area forward and places the cursor at the initial position of the first modifiable field in the current line.

When the cursor is on the bottom screen line of the scrolled area, or if the program specifies data to update the bottom line, the Scroll Forward function scrolls the top scrolled line of information off the screen, scrolls a new line of information into the bottom line of the scrolled area, and moves the intermediate scrolled lines upward. If the cursor is on the bottom line and if your pro- gram specifies values for the new line of information, they are displayed; otherwise, the default field values are displayed.

The function is invalid when the cursor is in a field that is not within a scrolled area.

2.3.6.6. Exiting Scrolled Area Backward

- Default VT100/VT200 Key Sequence: PF1 followed by the Uparrow
- Terminator Code

  Value: 4

  Symbol: FDV$K_FT_XBK

The Exit Scrolled Area Backward key is valid only when the cursor is in a field that is within a scrolled area. The function transfers control to your program unless your program is executing a GETAF call and the operator has not yet entered or changed a field value. Therefore, you can usually use the function in any way you want, and the effects that the operator sees depend on the next calls that your program issues.

The Form Driver processes the Exit Scrolled Area Backward key when you specify its code in the PFT call, except when your program is executing a GETAF call. The Form Driver moves the cursor to the initial position of the first modifiable field preceding the scrolled area.

The function is invalid when:

1. The cursor is in a field that is not within a scrolled area.

2. No modifiable field precedes the scrolled area.

2.3.6.7. Exiting Scrolled Area Forward

- Default VT100/VT200 Key Sequence: PF1 followed by the Downarrow

- Terminator Code

  Value: 5

  Symbol: FDV$K_FT_XFW

The Exit Scrolled Area Forward key is valid only when the cursor is in a field that is within a scrolled area. The function transfers control to your program unless your program is executing a GETAF call and the operator has not yet entered or changed a field value. Therefore, you can usually use the function in any way you want, and the effects that the operator sees depend on the next calls that your program issues.

The Form Driver processes the Exit Scrolled Area Forward key when you specify its code in the PFT call, except when your program is executing a GETAF call. The Form Driver moves the cursor to the initial position of the first modifiable field following the scrolled area.

The function is invalid when:

1. The cursor is in a field that is not within a scrolled area.

2. No modifiable field follows the scrolled area.

2.3.6.8. Illegal Terminator Interaction

If your program is sues the ILTRM call with an argument of 1, any terminator that is illegal in its current context — for example, a Previous Field terminator in the first field of a form — is converted to a special terminator code, treated as if it came from the pressing of a function key, and sent to a function key UAR, if any. (See the description of the ILTRM case in Chapter 5.)

The terminators that are affected are: NXT, ATB, PRV, XBK, XFW, SFW, and SBK.

The first five of these are illegal if there is no next or previous field. The last four are illegal if the current field is not in a scrolled area.

The illegal terminator symbols and values are:

FDV$K_FT_ILG_NXT= 11

FDV$K__FT_JLG_PRV = 12

FDV$K FT_JLG_ATB = 13

FDV$K_FT_ILG_XBK = 14

FDV$K_FT_ILG_XFW = 15

FDV$K__FT_ILG_SFW = 16

FDV$K FT_ILG_SBK = 17

2.3.7. Alternate Keypad Mode Terminators

Normally, the numeric and punctuation keys on the VT100 and LK201 keypads produce the same numbers and characters that the corresponding keyboard keys produce. Therefore, for many common applications, the operator can enter numeric data by using the keypad rather than the keyboard.

For special applications, you can set the VT100/VT200 to alternate keypad mode by issuing the SPADA call from your program or by entering the DCL command

```
$ SET TERMINAL/APPLICATION.KEYPAD
```

prior to running your application. You can then design the applications to use the numeric and punctuation keys on the keypad as field terminator keys.

The Form Driver then passes the alternate keypad mode terminators to the program immediately, regardless of whether the Response Required, Must Fill, and UAR requirements are satisfied for the form.

Table 2.6 and Table 2.7 include lists of the keypad keys that are affected by the alternate keypad setting and the code that is returned to your program for each key. Each character returned is the last character in the escape sequence generated by the key in alternate keypad mode.

2.4. Key Functions and Key Codes

This section provides a fuller explanation of the roles of function keys, key functions, and key codes and gives their values.

2.4.1. Form Driver Key Functions

Form Driver key functions are actions the Form Driver takes in response to special keystroke sequences. Key functions, values for the DFKBD call, and default key sequences are given in Table 2.4.

Table 2.4. Key Functions

Function Name	Description	Default VT100 Key sequence	DFKBD Value
FDV$K_KF_DLCHR	Delete character	DELETE	1
FDV$K_KF_CRSRT	Move cursor right	Rightarrow	2
FDV$K_JCF_CRSLF	Move cursor left	Leftarrow	3
FDV$K_KF_X>LFLD	Delete Field	LINEFEED	4
FDV$K_JCFJNS	Set Insert mode	PF1 PF3	5
FDV$K_JCF_OVR	Set Overstrike mode	PF3	6
FDV$K_KF_GOLD	Start Gold sequence	PF1	7

Function Name	Description	Default VT100 Key sequence	DFKBD Value
FDV$K_KF_RESET	Reset Gold sequence	PF1 DELETE	8
FDV$K_KF_RFRSH	Refresh screen	CTRL/R	9
FDV$K_KF_JHELP	Help	PF2	10
FDV$K_KF_NXT	Next field	TAB	11
FDV$K_KFJPRV	Previous field	BACKSPACE	12
FDV$iejCFJsrm	Form or field complete	RETURN ENTER	13
FDV$K_KF_SBK	Scroll backward	Uparrow	14
FDV$K_KFJ3FW	Scroll forward	Downarrow	15
FDV$K_KF_XBK	Exit scroll area backward	PF1 Uparrow	16
FDV$K_KF_XFW	Exit scroll area forward	PF1 Downarrow	17

The first nine key functions are called the editing key functions; they are handled internally by the Form Driver and are not returned to your program. Interpretation of the other key functions is context dependent they may be illegal, interpreted by the Form Driver, or returned to the calling program as terminators.

Terminators are values returned to the calling program to indicate how input requests were completed. That is, terminators are key codes with a context.

Key functions in the proper context can give rise to a terminator code not the same as the key function code. In addition, these key function terminators can be processed by the PFT call. Key codes that are not key functions are returned as terminators to the calling program.

For the DFKBD call, FDV$K_KF_NONE is 0, and FDV$K_KFJDFLT is -1.

2.4.2. Form Driver Key Codes

Form Driver key codes are 16-bit encodings of certain key sequences. These sequences, listed in Table 2.5, are control characters, legal ANSI escape sequences, and 2-stroke sequences beginning with the Gold key.

The definitions include the keystroke combinations interpreted by the Form Driver. Each combination has a coded value — an integer word associated with it. The key codes are used in two ways. The most common is as the terminator to a field; the Form Driver returns terminators, and these are the codes for those terminators. The other way is to use these values as the key codes passed to the DFKBD call.

All ASCII graphic characters are treated as data by the Form Driver and are not available for use as terminators, except as the second key in a Gold sequence. The remaining keystroke combinations are divided into three groups as follows:

1. Control keys, including the DELETE key

2. Escape sequences, including keypad application mode keys, cursor position keys, and program function keys

3. Gold sequences

2.4.2.1. Control Keys

Control characters — ASCII codes 0 to 31(decimal) — and the DELETE key — ASCII code 127(decimal) are not allowed as data in fields. A control character not defined as a key function is returned to the calling program as a terminator.

Control characters can be assigned to key functions, and several are assigned as defaults. The Form Driver key codes for control keys are listed in Table 2.5.

Note that the operating system may preempt the use of some control keys for example, CTRL/Y. Using control keys other than those assigned as the defaults may lead to unexpected results.

In Table 2.5, the first column contains the ASCII name for the control key, and the second column contains the ASCII value in decimal. Some of these codes for example, CR are produced by editing keys on the keyboard. Others are available only by holding down the CTRL key and pressing another key.

The third column contains the name of the key the operator presses while holding down the CTRL key. The fourth column contains the Form Driver key code the value to be used for the “defkbd” argument of the DFKBD call, or the value of the key as a terminator.

Note that the low-order byte of the key code is the 7-bit ASCII code for this key. For those keys used as part of the default Form Driver keyboard, the last column has the name of the associated key function. Some control keys have special meanings in terminals and cannot be used as terminators. This restriction is noted in the last column, although the absence of a note should not be taken as a guarantee that the key is available.

Table 2.5. Key Codes for Control Characters

ASCn Name	Value (Decimal)	CTRL/Key	Key Value (as terminator)	Default Assignment
NUL	00	@	1024 + 00	
SOH	01	A	1024 + 01	
STX	02	B	1024 + 02	
ETX	03	C	1024 + 03	
EOT	04	D	1024 + 04	
ENQ	05	E	1024 + 05	
ACK	06	F	1024 + 06	
BEL	07	G	1024 + 07	
BS	08	H	1024 + 08	FDV$KJKF_PRV
HT	09	I	1024 + 09	FDV$K_KF_NXT
LF	10	J	1024 + 10	FDV$K_KF_DLFLD
VT	11	K	1024 + 11	
FF	12	L	1024 + 12	
CR	13	M	1024 + 13	FDV$K_KF_NTR
SO	14	N	1024 + 14	
SI	15	O	1024 + 15	
DLE	16	P	1024 + 16	

ASCn Name	Value (Decimal)	CTRL/Key	Key Value (as terminator)	Default Assignment
DC1	17	Q	1024 + 17	
DC2	18	R	1024 + 18	FDV$K_KF_RFRSH
DC3	19	S	1024 + 19	
DC4	20	T	1024 + 20	
NAK	21	U	1024 + 21	
SYN	22	V	1024 + 22	
ETB	23	W	1024 + 23	FDV$K_KF_RFRSH
CAN	24	X	1024 + 24	
EM	25	Y	1024 + 25	
SUB	26	Z	1024 + 26	
ESC	27	[		Not available
FS	28	\	1024 + 28	
GS	29	]	1024 + 29	
RS	30	^	1024 + 30	
US	31	_	1024 + 31	
DEL	127		1024 + 127	FDV$K_KF_DLCHR

2.4.2.2. Escape Sequences

The second group of keys consists of the cursor control keys, the program function keys, and the application keypad keys (Table 2.6). Note that the cursor control keys and program function keys 1 to 3 are all assigned default Form Driver key functions. The key codes for the default editing functions are not returned to the program; instead, their functions are performed. For the terminator key functions, the context selected terminator code corresponding to the key function is returned.

Table 2.6. Key Codes for Escape Sequences

Key Code	Value	Name of Key	Default Key Function
FDV$K_JVR_UP	99	Uparrow	FDV$K_JCF_SBK
FDV$K_AKJDOWN	100	Downarrow	FDV$K_KF_SFW
FDV$K_jtfL_RIGHT	101	Rightarrow	FDV$K_JCF_CRSRT
FDV$K_AR_LEFT	102	Leftarrow	FDV$K_JCF_CRSLF
FDV$K_PF_1	103	VT100 PF1, VT52 Blue	FDV$K_KF_GOLD
FDV$K_PF_2	104	VT100 PF2,VT52 Red	FDV$K_JCF_HELP
FDV$K_PF_3	105	VT100 PF3, VT52 Gray	FDV$K_KF_OVR
FDV$K_PF_4	106	VT100 PF4	
FDV$K_KP_NTR	107	Alternate keypad ENTER	FDV$K_KF_NTR
FDV$K_KP_COM	108	Alternate keypad ,	
FDV$K_KP_HYP	109	Alternate keypad -	
FDV$K_KP_REP	110	Alternate keypad .	

Key Code	Value	Name of Key	Default Key Function
FDV$K_KP_0	112	Alternate keypad 0	
FDV$K_KP_1	113	Alternate keypad 1	
FDV$K_KP_2	114	Alternate keypad 2	
FDV$K_KP_3	115	Alternate keypad 3	
FDV$K_KP_4	116	Alternate keypad 4	
FDV$K_KP_5	117	Alternate keypad 5	
FDV$K_KP_6	118	Alternate keypad 6	
FDV$K_KP_7	119	Alternate keypad 7	
FDV$K_KP_8	120	Alternate keypad 8	
FDV$K_KP_9	121	Alternate keypad 9	
FDV$K_FK_E1	33	LK201 E1	
FDV$K_FK_E2	34	LK201 E2	
FDV$K_FK_E3	35	LK201 E3	
FDV$K_FK_E4	36	LK201 E4	
FDV$K_FK_E5	37	LK201 E5	
FDV$K_FK_E6	38	LK201 E6	
FDV$K_FK_F6	49	LK201 F6	
FDV$K_FK_F7	50	LK201 F7	
FDV$K_FK_F8	51	LK201 F8	
FDV$K_FK_F9	52	LK201 F9	
FDV$K_FK_F10	53	LK201 F10	
FDV$K_FK_F11	55	LK201 F11	
FDV$K_FK_F12	56	LK201 F12	FDV$K_KF_PRV
FDV$KJFLF13	57	LK201 F13	FDV$K_KF_DLFLD
FDV$K_FK_F14	58	LK201 F14	
FDV$K_FK_HELP	60	LK201 HELP	FDV$K_KF_HELP
FDV$K_FK_DO	61	LK201 DO	
FDV$K_FKJF17	63	LK201 F17	
FDV$K_FK_F18	64	LK201 F18	
FDV$K_FKJ19	65	LK201 F19	
FDV$K_FK_F20	66	LK201 F20	

2.4.2.3. Gold Sequences

The last group consists of sequences starting with the Gold key. Any key not preempted by the terminal can follow the Gold key. Pressing a Gold key more than once is equivalent to pressing it once. The operator can cancel a Gold key sequence by entering the FDV$K_KF_RESET is equivalent to the null sequence.

The key or escape sequence following the Gold key determines the key code as listed in Table 2.7. The sequences expected to be used most often are given names.

Table 2.7. Key Codes for Gold Escape Sequences

Key Code	Value	Key Sequence	Default Key Function
FDV$K_GAR_UP	227	Gold Uparrow	FDV$K_KF_XBK
FDV$K_GAR_DOWN	228	Gold Downarrow	FDV$K_KF_XFW
FDV$K_GAR_RIGHT	229	Gold Rightarrow	
FDV$K_GAR_LEFT	230	Gold Leftarrow	
FDV$K_GPF_1	231	Gold PF1 (VT100) Gold Blue (VT52)	FDV$K_KF_GOLD
FDV$K_GPF_2	232	Gold PF2 (VT100) Gold Red (VT52)	FDV$KJKF_HELP
FDV$K_GPF_3	233	Gold PF3 (VT100) Gold Gray (VT52)	FDV$K_KF_JNS
FDV$K_GPF_4	234	Gold PF4	
FDV$K_GKP_NTR	235	Gold Alt keypad ENTER	
FDV$K_GKP_COM	236	Gold Alt keypad ,	
FDV$K_GKP_HYP	237	Gold Alt keypad -	
FDV$KGKP_PER	238	Gold Alt keypad .	
FDV$K_GKP_0	240	Gold Alt keypad 0	
FDV$K_GKP_1	241	Gold Alt keypad 1	
FDV$K_GKP_2	424	Gold Alt keypad 2	
FDV$K_GKP_3	243	Gold Alt keypad 3	
FDV$K_GKP_4	244	Gold Alt keypad 4	
FDV$K_GKP_5	245	Gold Alt keypad 5	
FDV$K_GKP_6	246	Gold Alt keypad 6	
FDV$K_GKP_7	247	Gold Alt keypad 7	
FDV$K_GKP_8	248	Gold Alt keypad 8	
FDV$K_GKP_9	249	Gold Alt keypad 9	
FDV$K_GFK_E1	161	LK 201 GOLD E1	
FDV$K_GFK_E2	162	LK 201 GOLD E2	
FDV$K_GFK_E3	163	LK 201 GOLD E3	
FDV$K_GFK_E4	164	LK 201 GOLD E4	
FDV$K_GFK_E5	165	LK 201 GOLD E5	
FDV$K_GFK_£6	166	LK 201 GOLD E6	
FDV$K_GFK_F6	177	LK 201 GOLD F6	
FDV$K_GFK_F7	178	LK 201 GOLD F7	
FDV$K_GFK_J'8	179	LK 201 GOLD F8	
FDV$K_GFK_F9	180	LK 201 GOLD F9	
FDV$K_GFK_F10	181	LK 201 GOLD F10	

Key Code	Value	Key Sequence	Default Key Function
FDV$K_GFK_F11	183	LK 201 GOLD F11	
FDV$K_GFK_F12	184	LK 201 GOLD F12	
FDV$K_GFK_F13	185	LK 201 GOLD F13	
FDV$K_GFK_F14	186	LK 201 GOLD F14	
FDV$K_GFK_HELP	188	LK 201 HELP	FDV$K_KF_HELP
FDV$K_GFK_DO	189	LK201 DO	
FDV$K_FK_F17	191	LK201 GOLD F17	
FDV$K_GFK_F18	192	LK201 GOLD F18	
FDV$K_GFK_F19	193	LK201 GOLD F19	
FDV$K_GFK_F20	194	LK201 GOLD F20	

Note that the value for a Gold escape sequence is 128 plus the value for the escape sequence.

No symbols are defined for normal graphic or control keys preceded by the FDV$K_KF_GOLD key function because there are so many of them. Table 2.8 gives the key codes for these key sequences.

Table 2.8. Key Codes for Gold Sequence

Key	ASCII Value (Decimal)	Gold Key	Default Assignment
NUL	00	256 + 00	
SOH	01	256 + 01	
STX	02	256 + 02	
ETX	03	256 + 03	
EOT	04	256 + 04	
ENQ	05	256 + 05	
ACK	06	256 + 06	
BEL	07	256 + 07	
BS	08	256 + 08	
HT	09	256 + 09	
LF	10	256 + 10	
VT	11	256 + 11	
FF	12	256 + 12	
CR	13	256 + 13	
SO	14	256 + 14	
SI	15	256 + 15	
DLE	16	256 + 16	
DC1	17	256 + 17	
DC2	18	256 + 18	
DC3	19	256 + 19	
DC4	20	256 + 20	
NAK	21	256 + 21	

Key	ASCII Value (Decimal)	Gold Key	Default Assignment
SYN	22	256 + 22	
ETB	23	256 + 23	
CAN	24	256 + 24	
EM	25	256 + 25	
SUB	26	256 + 26	
ESC	27	**Not available**	
FS	28	256 + 28	
GS	29	256 + 29	
RS	30	256 + 30	
US	31	256 + 31	
SP	32	256 + 32	
.	33	256 + 33	
"	34	256 + 34	
#	35	256 + 35	
$	36	256 + 36	
%	37	256 + 37	
&	38	256 + 38	
'	39	256 + 39	
(	40	256 + 40	
)	41	256 + 41	
*	42	256 + 42	
+	43	256 + 43	
,	44	256 + 44	
-	45	256 + 45	
.	46	256 + 46	
/	47	256 + 47	
0	48	256 + 48	
1	49	256 + 49	
2	50	256 + 50	
3	51	256 + 51	
4	52	256 + 52	
5	53	256 + 53	
6	54	256 + 54	
7	55	256 + 55	
8	56	256 + 56	
9	57	256 + 57	
:	58	256 + 58	
;	59	256 + 59	

Key	ASCII Value (Decimal)	Gold Key	Default Assignment
<	60	256 + 60	
=	61	256 + 61	
>	63	256 + 62	
?	63	256 + 63	
@	64	256 + 64	
A	65	256 + 65	
B	66	256 + 66	
C	67	256 + 67	
D	68	256 + 68	
E	69	256 + 69	
F	70	256 + 70	
G	71	256 + 71	
H	72	256 + 72	
I	73	256 + 73	
J	74	256 + 74	
K	75	256 + 75	
L	76	256 + 76	
M	77	256 + 77	
N	78	256 + 78	
O	79	256 + 79	
P	80	256 + 80	
Q	81	256 + 81	
R	82	256 + 82	
S	83	256 + 83	
T	84	256 + 84	
U	85	256 + 85	
V	86	256 + 86	
W	87	256 + 87	
X	88	256 + 88	
Y	89	256 + 89	
Z	90	256 + 90	
[	91	256 + 91	
\	92	256 + 92	
]	93	256 + 93	
^	94	256 + 94	
_	95	256 + 95	
'	96	256 + 96	
a	97	256 + 97	

Key	ASCII Value (Decimal)	Gold Key	Default Assignment
b	98	256 + 98	
c	99	256 + 99	
d	100	256 + 100	
e	101	256 + 101	
f	102	256 + 102	
g	103	256 + 103	
h	104	256 + 104	
i	105	256 + 105	
j	106	256 + 106	
k	107	256 + 107	
l	108	256 + 108	
m	109	256 + 109	
n	110	256 + 110	
o	111	256 + 111	
p	112	256 + 112	
q	113	256 + 113	
r	114	256 + 114	
s	115	256 + 115	
t	116	256 + 116	
u	117	256 + 117	
v	118	256 + 118	
w	119	256 + 119	
x	120	256 + 120	
y	121	256 + 121	
z	122	256 + 122	
{	123	256 + 123	
|	124	256 + 124	
}	125	256 + 125	
~	126	256 + 126	
DEL	127	256 + 127	FDV$K_KF_EESET

Note

The key sequences listed below are reserved for future use by FMS. FMS may use them as default assignments in future versions. If you use any of them now, you may have to alter your programs later.

- Gold PF2
- Gold TAB
- Gold BACKSPACE

- Gold CTRL/R
- Gold CTRL/W
- Gold LINEFEED
- Gold RETURN
- Gold Rightarrow
- Gold Leftarrow

In addition, on the LK201 terminal keyboard, the following key sequences are reserved for future use by FMS:

- E1
- E2
- E3
- E4
- E5
- E6
- Gold E1
- Gold E2
- Gold E3
- Gold E4
- Gold E5
- Gold E6
- Gold TAB
- Gold F12
- Gold F13
- * Gold HELP

2.4.3. Defining Keys

The following example shows how to use the DFKBD call to switch the functions of the RETURN and the TAB keys. After this call is executed, RETURN or numeric keypad ENTER will mean Next Field (FDV$K_FT_NXT), and TAB will mean Enter Form (FDV$K_FT_NTR). The example is given in FORTRAN.

```
INTEGER TCA ( 3 )
INTEGER*2 KEYTABLE ( 4 ) / FDV$K_KF_NTR . 1033.
1    FD0*K_KF_NXT. 1037 /
CALL FDV$ATERM(ZDESCR(TCA).12.1)
```

```
CALL FDO$DFKBD(ZDESCR(KEYTABLE).2 )
```

2.5. Checking Call Status

To improve the effectiveness of VSI OpenVMS FMS applications and to reduce the time required for you to produce fully debugged applications, the Form Driver maintains the completion status of each call and provides five ways for you to obtain the status:

- Issuing the STAT call, which returns the Form Driver status code for the most recent call that was processed. The STAT call also returns the RMS system error code if a call fails because of an error in opening or reading a form library file or if other system problems are associated with terminal I/O. The status is returned as an integer longword as specified in an argument of the call.
- Issuing the SSRV call to establish either one or two global variables in your program to receive the FMS status after every FMS call.
- Issuing any call as a function returning a value. The returned status argument conforms to the VMS calling standard. The status is returned as an integer longword. Each language has a different way of obtaining the VMS return status immediately from a call. For example, in FORTRAN, the subroutine call is:

  ```
  CALL FDV*STAT(JSTAT JSTAT 2)
  ```

  The function value return call is:

  ```
  JSTAT=FDU*GET(FID,FUAL>TERM)
  ```

- Using the Form Driver Debug mode for displaying explicit messages about the status of erroneous calls for added support while an FMS application is being developed.
- Using the OpenVMS message facility with FMS. You can signal FMS errors from your program by using the standard VMS message facility. (See Table 2.9 for a list of the VSI OpenVMS FMS status returns.) You do this by using the LIB$SIGNAL call.

  When a VMS status is returned, it can be signaled as shown below:

  ```
  STATUS=FDY*LOPEN ('BADFILE')
  CALL LIB$SIGNAL (XYAL (STATUS))
  ```

Table 2.9 lists and describes the FMS status codes and the corresponding VMS status codes, which are global symbols. For FMS applications, the STAT call returns one of the listed numeric codes in the first of its two status arguments.

Two of the status conditions listed in Table 2.9 indicate an error in trying to open or read a form library file —code values FDV$_IOL and FDV$_IOR. In these two instances, the STAT call also returns, in the second status argument RMS system error codes that help to define the exact cause of the problem. For RMS errors, see the *OpenVMS System Messages and Recovery Procedures Reference Manual*.

In addition, error code FDV$_SYS indicates that the Form Driver has encountered an unexpected error in dealing with the operating system —terminal services, usually. The RMS code can be accessed to find the error status returned to the Form Driver, which may identify the problem —for example, network link lost.

Note that the status value FDV$_DLN —data specified too long for output — is reported only in Debug mode and is not returned to your program. Regardless of Form Driver support for Debug mode, the specified data is truncated when displayed, and the Form Driver completes the call in the normal way.

Table 2.9. FMS and VMS Status Codes

VMS Status Code	FMS Status Code	Meaning
FDV$J3UC	1	Successful completion of the call.
FDV$_INC	2	Form is incomplete after a PFT call.
FDV$JVIOD	3	Input successful. Field value in “fldval” has been modified by the operator.
FDV$_JMP	-2	Length specified in “wksp” descriptor is not large enough.
FDV$_FSP	-3	File specification in a LOPEN call was invalid.
FDV$_IOL	-4	Form Driver encountered an error while reading the form library. (It reads the form library to verify that the file is a form library file.)
FDV$_FLB	-5	Specified file was not a form library.
FDV$_ICH	-6	Channel specified was either in use or invalid.
FDV$_FCH	-7	Form was not resident, and when the Form Driver attempted to search for it in a form library, the current library channel was not open.
FDV$_FRM	-8	Form description is invalid.
FDV$_FNM	-9	Binary form description could not be found either in the form library, or in the list of memory-resident forms.
FDV$_LIN	-10	Line or portion of form lies outside the visible screen range.
FDV$_FLD	-11	Field does not exist, or index value is invalid for field.
FDV$_NOF	-12	Form contains no fields.
FDV$_DSP	-13	Form contains only Display Only fields, or the specified field is Display Only.
FDV$_NSC	-14	Field named is not a field in a scrolled area.
FDV$_DNM	-15	No Named Data is associated with the specified name or index.
FDV$_DLN	-16	Value argument supplied more data than was required, and some data was discarded.
FDV$_UTR	-17	Field terminator code is invalid.
FDV$JOR	-18	I/O error occurred while Form Driver was reading in a form from the form library. The I/O error code is recorded in the current state. You can obtain it by issuing the STAT call.

VMS Status Code	FMS Status Code	Meaning
FDV$JFN	-19	Field terminator code specified in the PFT call cannot be processed in the context indicated.
FDV$^ARG	-20	Incorrect number of arguments for call.
FDV$_ENI	-21	No workspace is defined.
FDV$_STR	-22	Value being returned is too large for the variable allocated for it.
FDV$JVM	-23	Not enough virtual memory could be allocated (either for the TCA or for the workspace).
FDV$JVM	-24	An error occurred in freeing virtual memory allocated to the application.
FDV$JTT	-25	Invalid terminal type.
FDV$JTCA	-26	Terminal Control Area is invalid or undefined.
FDV$_STA	-27	Size of specified TCA is too small.
FDV$_WID	-28	Form being displayed does not fit on the screen (132-column form on a VT52).
FDV$_NFL	-29	No form loaded into workspace.
FDV$_JBF	-30	Area not large enough to hold the form.
FDV$_NDS	-31	Form is marked as being not displayed, so no input is possible.
FDV$_UDP	-33	UAR depth was exceeded.
FDV$_UAR	-34	UAE returned an illegal code.
FDV$_UNF	-35	UAR was specified, but not found.
FDV$_CAN	-39	Call was terminated by a CANCL call.
FDV$_KIF	-40	Illegal key function was specified in DFKBD.
FDV$_JCEX	-41	Too many key codes were defined for some key function in DFKBD.
FDV$_KT	-42	Key code was given two key functions in DFKBD.
FDV$_KIL	-43	Illegal key code was given in DFKBD; that is, the key was not on the list in this chapter.
FDV$_TMO	-44	Operator took longer to respond than was allowed by the timeout value associated with the current terminal, for a GET-type call or WAIT call.
FDV$_LLI	-45	The Form Driver's internal buffer was not large enough to store the line image requested (in a RETFL call).The line image returned is truncated.

VMS Status Code	FMS Status Code	Meaning
FDV$_VAL	-47	The value of an argument is outside the allowed range.
FDV$JFU	-48	Illegal function while in currently active UAR.
FDV$_SYS	-49	Form Driver encountered system error response.
FDV$_INA	-50	Request information not available.

2.5.1. Debug Mode Support for Application Program Development

To use the Debug mode of the Form Driver, you need to make the following logical assignment at DCL level:

```
$ ASSIGN YES FDV $DEBUG
```

You can then run your application program without having to do any relinking. In Debug mode, the Form Driver reports explicit messages for status conditions of erroneous Form Driver calls. The Form Driver in Debug mode is useful during VSI OpenVMS FMS program development.

You can assign and deassign the FDV$DEBUG logical name during execution of your program, since the Form Driver checks the name upon each occurrence of an error.

Once you have debugged the program, you should deassign the Form Driver Debug mode. When your program is running, the operators do not see the Debug mode messages provided explicitly for program debug. See Section 2.5.2 for signaling the operator.

In Debug mode, the Form Driver signals you by ringing the terminal bell or reversing the screen video characteristics and displays a message on the bottom line of the screen for any of the error status conditions listed in Table 2.9. The *VSI FMS Utilities Reference Manual* lists the messages.

After displaying a Debug mode message, the Form Driver places the cursor in the lower right corner of the screen until you press the ENTER or the RETURN key, regardless of how you may have redefined any keys. This process prevents your program from clearing or overwriting a Debug mode mes sage before you have seen it. When you press the ENTER or the RETURN key, the bottom line is cleared, and your program resumes. It can then issue the PUTL call to display program-related messages on the bottom screen line.

The error code is returned to the calling program.

Because the Form Driver explicitly signals all call errors when Debug mode is in effect, you can use the Form Driver to debug your FMS program. Therefore, after debugging a program, you may choose not to test for certain errors that should not occur in a fully debugged application such errors as an incorrect field name or form name or an incorrect number of arguments in a call. The safest procedure is, of course, to check status after every Form Driver call.

Even in a finished FMS program, you should check, at a minimum, I/O errors after calls that:

- Open and close a form library file
- Display a form and must therefore read a form library file

- Solicit operator responses

Note

FMS does not interact with the VSI OpenVMS Debugger.

2.5.2. Signaling the Terminal Operator About Program Errors

Your program can signal an operator about a problem by issuing the PUTL call. Here is an example of an I/O error being reported. The following illustration shows one way you can use the PUTL call with the other status and error-checking features:

1. The program encounters an I/O error while trying to display a form.
2. The program detects the error by checking for a status of <0, using the STAT call. The call returns the error code —18 (FDV$_IOR) for an error in reading a form library file.
3. The program uses the status code as an index into a list of program-specific messages.
4. The program issues the PUTL call to display the message on the bottom screen line and a SIGOP call to get the operator's attention. The program then immediately issues the WAIT call to ensure that the message remains visible until the operator sees it and responds to it.

The calls are described in full in Chapter 5.

In BASIC:

```
100 CALL FDU$CDISP (FQRMNAME)
110 CALL FDV$STAT (FMSSTATUS)
200 IF FMSSTATUS < 0
THEN
CALL FDV$PUTL ('FORM ' + FORMNAME + ' NOT FOUND ')
CALL FDU$SIGOP
CALL FDU$WAIT
```

2.6. AST Considerations

The FMS Form Driver is optionally AST reentrant, but you must follow certain rules or risk severe problems, which the Form Driver cannot detect.

1. You must not attach the terminal (ATERM) with No AST support (default is AST support).
2. You can send output to the current form from an AST with no restrictions, although such output is more expensive in both time and characters sent to the screen. The reason for the added expense is that the Form Driver must always save and restore the video attributes and cursor position of the interrupted program.
3. You must not request input from an AST program.
4. You must not detach or switch a terminal or workspace or change a work-space involved in any current operation.

There may be additional restrictions on the use of FMS from ASTs, depending on the version of the operating system in use.

Chapter 3. Programming Techniques and Examples

Programming techniques are ways a programmer can exploit the capabilities of software inventively. In a new or greatly changed product, such ways are not likely to be immediately apparent to a new user.

Typically, techniques evolve naturally through normal use the user combines facilities in a certain way out of a need to accomplish a specific task, for example; or realizes that a facility meant to do one task is just the thing to solve some other kind of problem.

Descriptions of such techniques make up this chapter, along with programming examples. The following routines, written in various languages, are taken from the FMS Version 2 Sample Application Program (SAMP). The routines illustrate the value of using such capabilities as Named Data and the various kinds of user action routines (UARs). Their value is in preserving as much as possible, the independence of the application program that is, Named Data is associated with the form, and UARs are called from the Form Driver.

3.1. Scrolling

Because the Form Driver can store field values only for the fields that are on the terminal screen, your program must maintain all scrolled area field values that are not displayed; that is, all the values that are “above” and “below” each scrolled area. When your program scrolls the lines of a scrolled area upward or downward, the program must collect the lines of values scrolled out of the area, and display any line of values scrolled into the area. (See Chapter 2 for some discussion of this topic.)

3.1.1. Controlling Scrolled Areas

```
13000     DEF FN.VUEREG
13001     !+
13005     ! Subroutine OUEREG
13010     !   View the check register and scroll through it *
13015     !   Also display totals for current session.
13030     !
13035     ! Put UP resister form *
13040     CALL FDU$CDISP ('REGIST') \ C=FN.SROCHK
13072     !+
13075     ! Get number of lines in scroll area from form Named Data (item
 1).
13080     !-
13085     NSCROL* = ' ' i Pre-extend strins variable before call (BASIC
 only)*
13090     CALL FDU*RETDI (1Z* NSCROL* ) \ C=FN*SRVCHK
13095     NSCROLZ = VAL ( NSCROL* )
13100     !+
13105     ! Put lines from check register array into scrolled area*
13110     ! The window is initially from item 1 UP to item
13115     ! min(NSCROL* #LASTREGNUMZ), that is * UP to the size of the
 scrolled
13120     ! area or the size of the resister * whichever is less * Assume
 there
13125     ! is at least one line (the initial deposit)*
13130     ! -
```

```
13135     MINW 1ND0WZ = 1
13140     CALL FDU*PUTSC < 'NUMBER'* REGARRAY*(1) ) ! First line
13145     CURLINEZ = 1      \ Res item cursor is on
13150     WHILE ( CURLINEZ < LASTREGNUMZ AND CURLINEZ < NSCROLZ )
13155        CURLINEZ = CURLINEZ + 1
13160        CALL FDU*PFT < FDY*K^FT_SFW * 'NUMBER' )
13165        CALL FDV*PUTSC( 'NUMBER'* REGARRAY*( CURLINEZ ) )
13170     NEXT
13171     MAXWINDOWZ = CURLINEZ
13175     !+
13180     ! Get input from fake field of scrolled line and do what it says:
13185     !   Kpd . or RETURN/ENTER => return to menu
13190     !   UPARROW or TAB => scroll forward
13192     !   DOWNARROW or BACKSPACE => scroll backward
13195     !   all others => isnore
13200     ! Note that there is no form function Key UAR so this routine
13205     ! handles all terminators itself (by i*norin$ illegal ones).
13210     !-
13215     CALL FDV*GET < FAKE** TERMINATORZ * 'FAKE' )
13220     WHILE NOT (TERMINATORZ FDV*K_FT_NTR OR TERMINATORZ =
 FDU*K_KP_PER)
13225        IF TERMINATORZ = FDV*K_FT_SFW OR TERMINATORZ = FDU*K_FT_SNX
 THEN C=FN*SCRFWD
13235        IF TERMINATORZ = FDV*K_FT_SBK OR TERMINATORZ = FDU*K_FT_SPR
 THEN C=FN*SCRBAK
13245        CALL FDV*GET( FAKE* * TERMINATORZ * 'FAKE' )
13250     NEXT
13255     FNEND
```

3.1.2. Scrolling Forward

See also VUEREG routine at line 13000.

```
13500     DEF FN.SCRFWD
13501     !+
13505     ! Subroutine SCRFWO -- Scroll forward *
13510     !    CURLINEZ is the line in the register that the cursor is on.
13512     !    MINWINDOWZ and MAXWINDOWZ delimit the part of the register
13513     !    currently displayed in the scrolled area
13515     !-
13520
13525     !+
13530     ! If cursor is at the end of the register * report * and return
13535     !-
13540     IF CURLINEZ = LASTREGNUMZ THEN
             CALL FDU$PUTL( 'Last line of register' )
             FNEXIT
13545     !+
13550     ! If cursor not at the last line of a window * Just moue down
13555     ! If cursor is at the last line of a window *
13560     !     move window forward one line *
13565     !     write the new last line to the last line of the scrolled
 area
13567     ! Move current line pointer forward
13570     !-
13580     IF CURLINEZ. <> MAXWINDOWZ THEN
               CALL FDU*PFT < FDV$K_FT_SFW» 'NUMBER' )
          ELSE
```

```
               MINWINDOWZ = MINWINDOWZ + 1
               MAXWINDOWZ = MAXWINDOWZ + 1
               CALL FDO*PFT < FDV$K_FT_SFW * 'NUMBER'* REGARRAY$<
 MAXWINDOWZ ) )
13585     CURLINEZ = CURLINEZ + 1
13590     FNEND
13698
```

3.1.3. Scrolling Backward

See also VUEREG routine at line 13000.

```
13700     DEF FN.SCRBAK
13701     !+
13705     ! Subroutine SCRBAK -- Scroll bacKward
13710     !     CURLINEZ is the line in the register that the cursor is on
 *
13712     !     MINWINDOWZ and MAXWINDOWZ delimit the part of the register
13713     !     currently displayed in the scrolled area
13715     !-
13720
13725     !+
13730     ! If the cursor is at the beginning of the register * report *
 and return
13735     !-
13740     IF CURLINEZ = 1 THEN
               CALL FDV$PUTL( 'First line of register' )
               FNEXIT
13745     !+
13750     ! If cursor not at first line of the window * Just move UP
13755     ! If cursor is at first line of the window *
13760     !     move window back one line *
13765     !     write the new first line to the first line of the scrolled
 area
13767     ! Moue current line pointer back
13770     !-
13780     IF CURLINEZ < > MINWINDOWZ THEN
               CALL FDU$PFT( FDY$K_FT_SBK * 'NUMBER' )
          ELSE
               MINWINDOWZ = MINWINDOWZ - 1
               MAXWINDOWZ = MAXWINDOWZ - 1
               CALL FDU$PFT( FDO$K_FT_SBK, 'NUMBER'* REGARRAY
$( MINWINDOWZ ) )
13785     CURLINEZ = CURLINEZ - 1
13790     FNEND
```

3.2. Validating a One-Character Field- Using a UAR

Purpose

To check single-character fields for valid data input.

Description

Frequently when using forms as menus to select one of several options, a single-character field is used to enter a letter or number indicating the desired option. Although it is possible to have the application program test each of these fields separately to ensure that a valid choice has been entered, it is much more convenient to use a single UAR for this purpose.

Whenever a character is typed in a single-character field, the character is immediately checked against a list of permissible characters, and the operator is not allowed to proceed unless the character entered is found in the list.

Programming Technique

For you to use this technique for validating a single-character field, the associated data string of afield completion UAR must be set to contain a string of all the valid character responses. If space is a valid response, it must be embedded in the string, since trailing spaces are ignored.

When the data validation UAR is activated, it uses the RETCX call (Return Current Context) to recover the associated data string, and the RETFN and RET calls to get the field name and field value, respectively.

It then searches the associated data character string for the field value. If the value is found, the UAR returns successfully, allowing a GETAL in progress to proceed normally to the next field. If the value is not found, the UAR returns validation failure, which causes the Form Driver to signal the operator and stay in the current field.

This technique is also used throughout the Form Editor for validating the selections for form and field attributes.

Example

```
16010     !+
16015     ! VAL1D1
16017     !     UAR for field validation of any one character field * The
16020     !     UAR associated data has in it the lesal characters allowed
 *
16025     !     except that blanK is not all owed unless it appears before
16030     !     the first trailing blanK * For example an assoc * value
 strinS
16035     !     'air' implies that only the letters a * HI and r are
 allowed *
16040     !     A strins 'aqr' means that blank is acceptable in addition
16045     !     to a * and r * Note that this routine is case sensitive
16050     !     (that is * it checks for correct case) * You can set around
16055     !     case sensitivity by usins the force-uppercase field
 attribute *
16060     !     and puttins only capitals into the UAR associated value
16065     !     strins *
16070     !
16075     !     This routine can be used with any form and field since
16060     !     it determines the context for itself *
16095     !-
16088     !+
16083     DECLARE INTEGER CONSTANT &
```

```
                FDV)K_UVAI__SUC = 1000 » (Field completion success &
                FDV)K_UVAI__FAIL =1001 (Field completion failure
16090       ! Pre-extend the strinss into which FMS will return values
16095       !-
16096       FRMNAM) = SPACE) ( 31 )
16097       UARVAL) = SPACE) ( 80 )
16098       FLDNAME) = SPACE) ( 31 )
16099       FVALUE) = SPACE) ( 1 )
16105
16110       !+
16120       ! Retrieve context: we will ignore TCA address * WKSP address *
 FRMNAM) *
16125       !     CURPOS * FLDTRM * INSDVR * and HELPNUM using only UARVAL) »
 and
16127       !     only the initial * non-blank characters of it.
16130       ! Retrieve field name and index *
16135       ! Retrieve field value *
16140       CALL FDVSRETCXt TCAX » WKSP* » FRMNAM) * UARVAL) * CURPOSX .
 FLDTRMZ » INSOVRX , HELPNUMZ )
16142       UARVAL) = TRM) ( UARVAL) )
16145       CALL FDVSRETFNC FLDNAME) * FINDEXX )
16150       CALL FDV)RET ( FVALUE) * FLDNAME) * FINDEXX )
16160
16165       !+
16170       ! To be valid * FVALUE) must occur in the strinS UARVAL)
16175       !-
16165       IF POS ( UARVAL) * FVALUE) * 1 ) > 0 THEN
                VALID1 = FDV)K_UVAI__SUC \ Suecess
            ELSE
                CALL FDV)PUTL < 'Illegal value' >
                VAL1D1 = FDV)K_UVAI__FAIL
16210       FUNCTIONEND
```

3.3. Producing Hard Copy - Using Named Data

Purpose

To produce a printable image of a form or portion of a form.

Description

A common application requirement is to produce a printable copy of a form on the screen. Complications arise if only part of the form is to be printed or if only one form out of a set of multiple forms on the screen is to be printed. The program could select particular lines to be printed, but doing so would destroy some of the program/form independence that is one of the chief virtues of FMS. Changes to the layout of a form might then require the program to be changed when the changes would be only cosmetic and should not be affecting the program.

Programming Technique

The RETFL call returns the printable image of an individual line on the screen. These line images are suitable for writing directly to a line printer or to a file for later printing. If the entire screen is to be printed, a program loop requesting lines one through twenty-three will produce the twenty-three line images.

When only part of the screen is required, the difficulty is in knowing which lines to ask for in the RETFL call. A common technique is to store the range of the lines to be printed with the form itself. At form creation time the form designer knows what lines are to be printed and puts the numbers of the first and last lines to be printed in the form's Named Data. The program accesses the Named Data to find the first and last lines to print and uses them as limits on the program loop calling RETFL. Then, if the form layout is ever changed so that a different range of lines should be printed, the form designer changes the Named Data and the unchanged program still produces the desired result.

Example

The following extract from the FMS Sample Application program shows this technique in action. The form has two Named Data items with names FIRST and LAST indicating the first and last lines to print. Each item has two characters representing the number. The program reads those Named Data items, which are character strings, converts them to numbers for internal use, and uses them as limits on a loop that includes a call on RETFL and a statement that writes the line images to a data file.

The following segment is shown in FORTRAN. See the *VSI FMS Language Interface Manual* for the equivalent code segment for any other languages supported by FMS. That manual also has descriptions of the CHECK form.

```
          SUBROUTINE PRINT_THE_CHECK
C             Print the check into the file SAMPCH * DAT
          CHARACTER*BO LINE
          CHARACTERS FIRSTL *
          1          LASTL
          INTEGER    FIRST_LINE_NUMBER *
          1          LAST_LINE_NUMBER ,
          2          I »
          3          LINELENGTH
C         Open check writing file * Note there's a new version
C         for every check *
C
          OPEN(UNIT=2, FILE='SAMPCH.DAT'» STATUS='NEW',
          1          CARRIAGECONTROL='LIST', RECORDSI2E=80 )

C             Get the top and bottom lines of the check from the named data
C             (first two characters)*

          CALL FDV$RETDN( 'FIRST', FIRSTL )
          CALL CHECK-FMSSTATUS()
          CALL FDV$RETDN < 'LAST', LASTL )
          CALL CHECK-FMSSTATUS()

C             Get lines from form.
C             Write to file *

          READ (FIRSTL, '(12)') FIRST-LINE-NUMBER
          READ (LASTL, '(12)') LAST_LINE-NUMBER

          DO I = FIRST_LINE_NUMBER, LAST_LINE_NUMBER
             CALL FDV*RETFL ( I, LINE, LINELENGTH)
             WRITE <2»'(A)') LINE(1:LINELENGTH)
          ENDDO
          CALL FDV$PUTL( 'Check written to file')
          CLOSE (2)
```

```
            END
D
```

3.4. Storing Message Text - Using Named Data

Purpose

Keep operator message texts independent of your program.

Description

Messages to the terminal operator are often changed during development of an application. To keep the messages independent of the peculiarities of a particular programming language, you can use files and modules that contain only message text.

When a form changes, it is often necessary to change the message file also. Even after program development, convenient change of operator messages is desirable. A program product may need to change both forms and message files to be tailored for a new customer. A program that is used by operators who speak different languages must maintain different form and message files for each language.

Programming Technique

One way of simplifying control of operator interaction is to keep all text that is presented to an operator in an FMS form. In the case of a multilingual environment, the application designer can develop forms that all request the same information, but that have background text appropriate to the language (for example, German and French).

All forms for a single language are collected in a form library. There may be several such libraries, each containing forms having identical names, with the only difference being the background text. The first form that the application displays is a menu form requesting the operator to select a language. The application opens the appropriate form library according to the language selected.

Thereafter, when a form is called from the library (for example, with a CDISP call), the named form is read from the library and displayed on the screen, presenting text in the operator's language. The application doesn't need to be concerned with the language at that point, since the previously made choice controls the form displayed now.

Since the application must occasionally display additional messages (perhaps by means of the PUTL call to the bottom line of the screen), it is consistent to store the text of those messages in the Named Data of the form. Either the name or index of the Named Data item can be used as an identifier to retrieve the text from the form before sending it to the operator.

In the multilingual environment, these messages can be in the selected language. Even for single language environments, storing the message with the form makes sense, since the messages often relate to the form. The form designer can change the form and the related messages in the same place, saving time and the usual confusion when separate files must be used together.

Example

The Sample Application shows one example of how to use Named Data to store message text in the DEPOSIT form. After the operator has entered checking account deposit information, SAMP wishes to display a message to the effect that the operation is done and the operator should press the RETURN key

to continue. The form used for deposit entry has such a message stored in the Named Data item with the name DONE.

The following extract from the FORTRAN SAMP shows how this can be done. Consult the *VSI FMS Language Interface Manual* to see the form definition for DEPOSIT and the SAMP in any of several languages.

```
CHARACTER*80 DONE
CALL FDY$RETDN < 'DONE ' , DONE)
CALL FDO$PUTL (DONE)
CALL FDY*WAIT
```

3.5. Converting Function Keys to Field Entry

Purpose

Provide an easy way of accepting either function keys or text in a field to select an option from a menu.

Description

In menu entry forms it is often desirable to allow the operator to enter the choice in one of two ways: enter the option followed by a terminator, or enter a function key. It is often inconvenient to implement such a design, since it involves two sets of validations instead of just one (making sure the text is correct, or if that is blank, making sure the function key is correct). Nonetheless, the convenience of a single keystroke menu response makes it worth considering.

Programming Technique

A function key UAR can convert a function key to the text string it is equivalent to. The UAR then outputs that text string to the menu's choice field as if the operator had entered the text. The return code for the UAR tells the Form Driver to process the field as if the operator had pressed the RETURN key instead of a function key. The Form Driver then calls any field completion UARs or returns control directly to the calling program, which only has to look at the text, regardless of whether it has been entered from the keyboard or by means of a function key.

Example

The Sample Application's MENU form has a function key UAR that converts six function keys into the text strings "1", "2", "3", "4", or "5". The function keys accepted are all on the application keypad. The key 1 and the key period (.) are both converted to the string "1"; the key 2 is converted to the string "2"; keys 3, 4, and 5 are converted to strings "3", "4", and "5". All other function keys are rejected. While this is a specific UAR, more general UARs to do this conversion can be written.

The following extract from the COBOL SAMP shows how this can be done. Consult the *VSI FMS Language Interface Manual* to see the form definition for DEPOSIT and the SAMP in any of several languages.

```
IDENTIFICATION DIVISION.
PROGRAM-ID. TAKE15 INITIAL *
*************************************************************
*  Function Key User Action Routine for the MENU form of SAMP  *
*  Convert Keypad 1-5 into field values 1-5 »                  *
*  Convert Keypad period into field value 1 »                  *
```

```
*  Reject all other function Keys with error message.          *
*************************************************************
DATA DIVISION *
WORKING-STORAGE SECTION *
             COPY " FDVDEF " *
             COPY " SAMPCOB" *
             COPY " SMPCOBUAR" *
*
*   Declarations specific to this UAR *
*
01           FIELD_VALUE PIC X ( 1 ) VALUE SPACE *
01           ILLEGAL__FUNC_KEY_MSG PIC X ( 20 )
                    VALUE "Illegal function Key" .
PROCEDURE DIVISION   GIVING RETURN_STATUS*
*-
* Retrieve contexts ignore all but TERMINATOR
*-
  CALL " FDV*RETCX " USING BY REFERENCE ADDRESS_TCA »
      BY REFERENCE   ADDRESS_WKSP #
      BY DESCRIPTOR  FORM_NAME t
      BY DESCRIPTOR  UAR_DATA »
      BY REFERENCE   CURSOR_POSITION #
      BY REFERENCE   TERMINATOR #
      BY REFERENCE   INSOVR_STATUS #
      BY REFERENCE   HELP_STRIKES .
#+
* Do the conversion # displaying the value converted if found*
* Reject if not one of the expected terminators*
#-
  EVALUATE TERMINATOR
      WHEN FDV$K_KP_1    MOVE "1" TO FIELD_VALUE
      WHEN FDV$K_KP_2    MOVE "2" TO FIELD_VALUE
      WHEN FDV*K_KP_3    MOVE "3" TO FIELD_VALUE
      WHEN FDV$K_KP_4    MOVE "4" TO FIELD_VALUE
      WHEN FDV$K_KP_5    MOVE "5" TO FIELD_VALUE
      WHEN FDV$K_KP_PER  MOVE mlm TO FIELD_VALUE
  END-EVALUATE.
  IF FIELD_VALUE = SPACE THEN
      CALL "FDV$PUTL" USING
    BY DESCRIPTOR ILLEGAL_FUNC_KEY_MSG
    CALL "FDV$SIGOP"
    Just ignore it now *
    MOVE FDV$K_UKEY_SUC TO RETURN_STATUS
  ELSE
    CALL "FDV$PUT" USING BY DESCRIPTOR FIELD_VALUE
      BY DESCRIPTOR N_MENU_OPTION
    Treat as if it is RETURN *
    MOVE FDV$K_UKEY_NTR TO RETURN_STATUS
  END-1F *
  EXIT PROGRAM *
END PROGRAM TAKE15 *
```

3.6. Filter for Function Keys

Purpose

Allow only certain function keys to be returned to the program.

Description

FMS defines a great many function keys (control keys, Gold sequences, terminal function keys, alternate keypad keys), but most applications only need a few keys active during the processing of a particular form. Other keys can be rejected or ignored. On return from a GET-type call the application can determine whether the terminator is legal, but this is such a common requirement that a general purpose routine can save a lot of trouble.

Programming Technique

Define a general purpose function key UAR for the form, that allows only certain function keys to be returned to the program. One way of doing this is to have the UAR associated data be a string that has the keycodes of the legal function keys.

The form designer then specifies the UAR in the Form Phase of the Form Editor or in the FORM statement of the Form Language, with an associated data value representing just those function keys that are legal. (About twenty keys could be specified in this fashion, more than an operator can usually deal with.) The function key UAR reads the associated data string and compares the values found there to the keycode received, rejecting the key if no match is found.

Alternatively, if you want to change the legal function keys more often than you switch the form, or if you wish to have more keys than can be listed in the eighty bytes of the UAR data, define a COMMON area with an array containing the legal keycodes, and a variable specifying how many different keycodes are currently in the array.

The application program updates the array and the counter variable whenever it determines that a different set of functions is legal. A function key UAR can access the array in COMMON, comparing the key code received against the legal values, and returning success to the Form Driver only if a match is found.

Example

The Sample Application has a function key UAR called PASSKY that is used on each of the data entry forms in SAMP. PASSKY implements the first of the suggestions above — the UAR associated data string has the legal keycode values. PASSKY is given below in its PASCAL version. Consult the *VSI FMS Language Interface Manual* to find PASSKY in any of several languages. You can also find references to the SAMP forms that use PASSKY.

(It is possible to write a more efficient implementation of PASSKY than is shown here. Instead of converting each of the character strings in the UAR data string to binary and then comparing the binary number to the terminator, you can convert the terminator to a four character ASCII string (with leading zeros) and then use a string function to see if it appears in the UAR data string. Each string in the UAR data would have to be four characters long, with leading zeros, for this to work.)

```
FUNCTION PASSKYi
C  General function Key uar to pass only those from the (small)
   list in the uar associated value strips and reject all
   others * The list is of the form:
          n <oneblank> n <oneblanK> *** n <manvblanKs>
   For example the string '110 112' would accept Keypad period
   and keypad zero but no other function Keys* >
```

```
LABEL   1000;
VAR     Nexttrm:   INTEGER$
        NonBlank:  INTEGER5
        NextBlank: INTEGER*

BEGIN

< Retrieve context; we will isnore TCA address * WKSP address*
             FRMNAM * INSOVR * and CURPOS * usin* only FLDTRM and
             UARVAL* >
FDV$RETCX (TCA : = Tea * WKSP : = Workspace * FRMNAM := Frmnam *
      UARVAL := Uarval * CURPOS ; = Curpos * FLDTRM := Fldtrm #
      INSOVR := Irsovr * HLPNUM : = Hlpnum ) *
i Break UP the list into numbers * Check each asainst the
  terminator * If terminator found in list * return success* >

Nonb1anK := 15        < Besfinnintf of strinSl
WHILE (UarvalCNonblankl <> ' ') AND (Nonblank <= 80 ) DO
      BEGIN
      Nextblank := INDEX (SUBSTR (Uarual * Nonblank *
                          LENGTH (Uarval) - Nonblank + 1) » ' ')
      IF Next blank = 0
      THEN Nextblank := 80
      ELSE Nextblank := Nextblank + Nonblank - 15
      READY (SUBSTR (Uarval » Nonblank * Nextb1ank - Nonb1ank) »
                     Nexttrm) 5
      IF Fldtrm = Nexttrm
      THEN
                  BEGIN
                  PASSKY := FDV$K_UKEY_TRM5 (Pass Key to application!)
                  GOTO 1000 *
                  END ;
      Nonblank := Next blank + 1 *
      END *
PASSKY := FDY$K_UKEY_ERR5   C Let FDD do the beepins}
1000s END *
```

3.7. Range Checks for Fields

Purpose

Ensure that a field contains values only in a particular range.

Description

Many fields can contain numeric values only in certain ranges, which are known ahead of time and which do not need to change dynamically. For example, there may be a minimum order on certain items, or only certain temperature ranges may be possible in a laboratory environment. While it is possible to check these values in the application, it is more convenient to define a general purpose field completion UAR.

UARs are particularly useful because the main logic of the application program does not then have to concern itself with validity checks of this sort. The validity checks still have to be made, but they are made in a modular fashion in a subroutine that does not clutter up the main line, and that is usually more concerned with relationships between the entered data and a database or real time process.

Programming Technique

A range checking UAR can be specified for each field that requires range checking. The lower and upper bounds for the field values are specified in the UAR associated data, separated by a comma. If no lower or upper bound is given, then no check for the bound is made, allowing ranges with open bounds on one end. The UAR data can also contain an error message to be issued in case of failure to satisfy the validity check.

Just putting a UAR on a field doesn't always mean that the UAR is called.

There are two conditions (other than error conditions, cancellation of the call, and field timeout) under which the Form Driver does not call a UAR for afield the field was terminated by the Previous Field key, or the field was terminated by a user function key.

In either of these situations the program must realize that the field may have invalid data. The program can take steps to guarantee that the UARs for the field get called so that its validity is assured. The program may refuse to accept such a terminator, and reestablish the read on the field or the form as a whole.

The program may call the PFT routine with the Enter Form terminator (FDV$K FT NTR). The Form Driver then checks all nonscrolled fields for validity, calling their UARs and returning a special status code to the program if any field fails to pass all checks. The program can reissue the read for the failed field and continue until the PFT routine returns success.

Example

The Sample Application program has a function, RANGE, that is called as a field completion UAR. The BASIC version of RANGE is given below. Consult the *VSI FMS Language Interface Manual* for examples of RANGE in other languages. Refer to the CHECK form in that manual for the field that uses this UAR.

Note that RANGE is not completely robust with respect to the UAR associated data string. A string that contains illegal numeric values on either side of the comma will cause problems. If your program is debugged, RANGE causes no problems, but a more general function would have to have some method of either checking for valid numbers, or a method of recovering from errors. The RANGE function in SAMP is used for integer values. Some of the language implementations actually allow decimal numbers because of the particular conversion functions used (for example, BASIC).

You can make RANGE more efficient if you require that the numbers in the UAR string be fixed format instead of free format. For example, the lower bound might occupy string positions 1-10; the upper bound, positions 11-20; and the error message, positions 21-80. The advantage for RANGE is in not having to scan for the numbers. The disadvantage, of course, would be for the form designer — fixed format input is inconvenient and subject to error.

```
FUNCTION INTEGER RANGE
!*********************************************************************
! General purpose UAR to check the range of any numeric item *
! associated UAR data must haue one of the four forms:
! L»U<space>{«essage>
! #U<spaceXmessage>
! L#<spaceXmessage>
! »<spaceXmessage>
! where L is lower bound * U is upper bound * and {message} is an
! optional error message in case the field ualue is out of bounds*
! If one of the bounds isn't siven» it isn't checked for *
! bound is given * nothing * is checked * everything succeeds *
```

```
! UAR ualue doesn't haue a comma * a FDU$_UAR error message is returned
! to the calling program by the FDU so the form designer has to go
! back and do it right. If no {message} is given # a simple
! " out of range U:L " message is giuen to the hapless operator*
!
! This UAR can work with any form and numeric field since it gets
! context itself * Care must be taken with field using field marker
! periods since those periods are not returned to the program *
! ********************************************************************
DECLARE INTEGER CONSTANT
                FDO$K_UOAL_SUC = 1000 »  !Field completion success &
                FDU$K^UUAL_FAIL = 1001   !Field completion failure
!+
! Pre-extend the strings into which FMS will return values.
! Get context which yields as sociated data ualue (ignore other stuff)*
! Get current field name and index*
! Get field ualue*
!-
FRMNAM* = SPACE*(31)
UARUAL* = S PACE*(80)
NAME$ = SPACE*(31)
NUMBER* = 5PACE*(132)
CALL FDU*RETCX (TCA7 » MKSPZ* FRMNAM* » UARUAL* * CURPOSZ* FLDTRM7* &
                        INSOORI * HELPNUM7. )
CALL FD0*RETFN (NAME* * INDEX!)
CALL FDU*RET (NUMBER* * NAME* # INDEX!)
NUMBER = UAL (NUMBER* )
!+
! Find comma and blank delimiters *
! CheCK for lower bound *
! _
COMMA* = P0S( UARUAL$ * 't' > 1 )
BLANK* = POS( UARUAL$t SPACE*( 1 ) > COMMA* + 1)
IF COMMA* = 0 THEN
   RANGE = 0 ! Illegal UARUAL string * FDU returns error
   FUNCTIONEXIT
IF COMMA* < > 1 THEN
   IF NUMBER < VAL ( SEG* (UARMAL$ * It COMMA* - 1)) THEN 20300
!+
! CheCK for upper bound
!-
IF BLANK* <> COMMA* + 1 THEN
      IF NUMBER > UAL(SEG*<UARMAL$ * COMMA* + It BLANK* - 1 >> THEN 20300
!+
! Passed both tests successfully t return success for UAR value
!
RANGE = FDO*K_UVAL_SUC
FUNCTIONEXIT
!+
! Error in one of the bounds *
! Give error message: either from the UARVAL or make one UP*
!-
IF SEG*( UARUAL* > BLANK* + 1 » BLANK* + 1) 0 SPACE$(1) THEN
   CALL FDU$PUTLt SEG$< UAROAL$t BLANK* + 1 » BO))
ELSE
   CALL FDU$PUTL( 'Field value out of bounds * Must be in ranSe » i + &:
                   SEG$( UARUAL$t 1 » BLANK* - 1) + / H. ' )
CALL FDU$SIGOP      ! Beept too *
```

```
RANGE = FDV$K_UUAI__FAIL
FUNCTIONEND
```

3.8. Simulating the GETAL Call

Purpose

General purpose structure for getting input from all fields in a form.

Description

One of the advantages of the GETAL call is that it allows the operator to progress through the form with the Next Field and Previous Field keys, filling in fields in any order (subject to Response Required and Must Fill attributes, and UAR validations). Only when the operator presses a function key or the Enter Form key does the Form Driver return control to the application; and if the Enter Form key is pressed, the Form Driver does not return control if any field fails validation.

Even though a program requires input from each input field on a form, there are many situations in which the GETAL call is inappropriate. GETAL does not access scrolled fields, and in some cases the mainline program needs to regain control after each field is entered.

The restriction on no scrolled fields is put on GETAL because the program must know at all times what the current scrolled line is in a scrolled area. If GETAL were to accept input from scrolled lines, and were to allow the user to scroll down, the program's knowledge of the current scrolled line would not match the Form Driver's knowledge. The reason for this mismatch is that there is no way in FMS for the Form Driver to tell the application the number of the current scrolled line, and no way to inform the application if the whole area is to be scrolled.

The only way the current scrolled line is changed is by the application's requesting a change (by means of the PFT call with the SFW, SBK, SNX, and SPR terminators; or by means of PUTDA's restoring the current scrolled lines of all areas to one).Since the application controls scrolling it can always know the current line.

A mainline program might wish to regain control after every field to achieve special effects. These effects might not be appropriate for UAR processing for a variety of reasons. The restrictions on UAR processing may be too severe for the effect. For example, the effect might be to skip over some field if some other field has a particular value, but a UAR cannot change the current field; or, the language being used does not support external routines.

Although these are good reasons not to call GETAL, it is not advisable to give up the operator's apparent freedom of order entry afforded by GETAL.

Programming Technique

It is possible to give the operator the same apparent freedom of control and still have control return to the application for every field. The general idea is:

1. Perform a GET for the first (or any other field).
2. Do the special processing for the named field after control returns to the program.
3. If the terminator specifies scrolling and you are in a scrolled area, update your data pointers for that scrolled area. (The Form Driver does not normally return scrolling terminators if the field is not in a scrolled area.)

4. Call PFT, specifying the field name used in the GET, and the terminator that was returned by GET. Ask that the new current field name be returned by PFT. This step requests that the action expected by the operator be carried out, at least in the internal memory of the computer (changing the current field). Note that neither the cursor nor the screen changes because of the PFT call.

5. Inspect the return from the PFT call. It can be one of four values:

   - FDV$_SUC: Success; the field name returned by PFT is the new current field name. If the terminator was not FDV$K_FT_NTR, your program can continue asking for input using the field name returned. If the terminator was FDV$K_FT_NTR, then the input is finished.

   - FDV$_INC: The field terminator was FDV$_FT_NTR and some field did not pass all the validation criteria. The current field is set to the first such field and is returned to your program. Your program can continue asking for input using the field name returned.

   - FDV$_UTR: The field terminator was a function key and not a terminator known to FMS. The current field is not changed and its name is returned to your program. You then choose what your program does depending on the function key. You may choose to continue input from the current field, which was returned to you.

   - FDV$_IFN: The terminator requests an illegal function in the current context (for example, Next Field at the end of the form). The only way this can be returned is if you have changed Supervisor Only mode since the GET statement. The GET call does not return a terminator that is illegal in the current context, so it must be that the context has changed.

     For example, at the time of the GET call, Supervisor Only mode was off and there was a Supervisor Only field following the current field, making the Next Field terminator legal. If you turn on Supervisor Only mode and there are only Supervisor Only fields following the current field, the Next Field terminator is now illegal. If you change the context you must decide what to do next. (Note that turning on Supervisor Only mode may also change the validity of the current field, since it may no longer be a modifiable field.)

6. Note that for the first three (normal) cases, your program may elect to continue input with the field that was returned to you by the PFT call. Your program can loop back to use that field name in the next GET call. Using the new current field name in the GET call makes it appear to the operator as if the cursor has moved in response to the terminator entered, which it indeed has, but only after the program has requested such movement.

When the FDV$_FT_NTR terminator is entered and PFT returns success for it, your program knows that all the nonscrolled fields have been entered correctly. You can be assured that all scrolled fields that were entered with any terminator other than FDV$_FT_PRV, FDV$_FT_SPR, or a function key are correct. That is, the scrolled line fields were validated up to the farthest point on the line reached.

Note that this technique works for any form since it does not need to know the field names. Of course, if you wish to do special processing for particular fields, you need the field names.

Example

The following code is extracted (in slightly modified form) from the Sample Application. The technique is used in SAMP only for illustrative purposes (there is no special processing done for a field). The difference between the SAMP code and that given here is that the SAMP code does a GETAF for most of the fields so that it regains control only after a field changes (except the first). The code listed below regains control after every field. Depending on the needs of your program, you may choose to do one or the other. The code below also differs from the SAMP in the way the call status is obtained after the PFT call.

The code segment below is for PL/I. Consult the *VSI FMS Language Interface Manual* for the SAMP program using GETAF in your favorite language.

```
SIMULATE: PROCEDURE
DCL    FIELDNAME CHAR (31)* / Name of field*/
       FIELDINDEX FIXED BIN (31) » /*Index of field*/
       FIELDVALUE CHAR (80)? /*Value of field*/

FIELDNAME = '*';    /* Identifies first field inform */
DO WHILE ('1'B) ?
   CALL FDV$GET < FIELDVALUE # TERMINATOR # FIELDNAME #
                  FIELDINDEX )

   /*
   /* Do any special processing for field FIELDNAME here#
   /* * « *
   /* Go to next or previous field or leave form
   /*                        */
   CALL FDV$PFT (TERMINATOR # # t FIELDNAME # FIELDINDEX ) ?
   /*
   /* If status is error # then PFT failed because terminator
   /* was a Keypad Key # which means return to caller #
   /*                        */
   CALL FDV$STAT (FMSSTATUS) ?
   IF FMS STATUS < 0 THEN RETURN 5
   IF TERMINATOR = FDV$K_FT_NTR
   THEN IF FMS STATUS * = 2
           THEN RETURN?
           ELSE DO?
                 CALL FDV$PUTL( ' INPUT REQUIRED ' ) ?
                 CALL FDV$BELL ?
                END ?
   /* LOOP # usin* new field name */
END ?
END SIMULATE 5
```

3.9. Reducing Display - Times for Forms

Purpose

Reduce the time to display a form and application supplied data.

Description

It is often the case that you will display a form (for example, with DISP) and then immediately output data to all the form's fields. This results in the fields on the forms being written twice once as the result of the DISP call outputting the default values, and once as the result of PUT calls or the PUTAL call. On a form with many fields this is a noticeable delay, especially on low-speed lines, and is distracting.

Programming Technique

Instead of using a DISP call to load the form into a workspace and display the form, and then using PUT calls, perform the following sequence:

1. Load the form into the workspace with LOAD (which does not display the form).

2. Perform the initial PUTs or PUTAL. This changes the workspace but does not output anything to the screen.

3. Call DISPW to display the form.

Note that this requires only one more Form Driver call (the DISPW call). It produces a significant reduction in terminal output for some forms.

3.10. Checking Status - Three Methods

Purpose

Determine result of Form Driver calls.

Description

Three methods of checking status of Form Driver calls are discussed in this section calling the Form Driver as a function, calling STAT, and setting up status recording variables with SSRV. The first method is compatible with the VMS calling standard. You would use the other two methods if you wish to be able to report a secondary status when an error is detected. (See Section 2.5.)

Programming Technique

In some situations, an error status does not indicate malfunctioning of your program but rather a special situation. For example, your program might be using Named Data as the list of valid table entries for a field. Asking the Form Driver for the data associated with a Named Data name is the only way of determining whether that Named Data item exists. If the Form Driver returns an error, your program may use the existence of the error as information itself —that the table entry is not valid.

In these situations, an error response from the Form Driver is not unexpected and does not represent a threat to the continuing execution of the program. Any of the three error determination methods is useful in this situation, since only the general status is needed. Using STAT is slightly more expensive since it requires an extra Form Driver call to obtain the call status.

In checking for unexpected error situations, no matter which technique you use, it is convenient to set up a subroutine to interpret the status of a call and take appropriate action. The subroutine checks the status and returns if it indicates success. If there is an unexpected error, the subroutine reports the error and stops the process in some fashion you determine. If the error is one of the types that has a secondary error status associated with it (FDV$_IOL, FDV$_IOR, or FDV$_SYS) you may want to report the secondary status also.

Example

A program that checks the Form Driver's function value return for an error can use the status in a call to the VMS RTL signaling routines, so that the message associated with the error is printed. It is still useful to do this in a subroutine since the subroutine then obtains the secondary status and also signals that. The program call on the subroutine would be used in the following manner (in PL/I):

```
CALLFMS = FD0*AWKSP < CHECKWKSR » 2000 )i
CALL CHKSTA( CALLFMS )i
```

The subroutine could be written as follows:

```
CHKSTA: PROCEDURE < FMS_OMS_STATUS )
/****_*************************************************
```

```
/* Subroutine CHKSTA
/* Check FMS status by looking at the parameter which is
/* a MMS status variable * If there is an error # detach
/* the terminal to clean UP screen and then
/* output the error by sitfnallintf *
/*************************************************/
DCL FMS_YMS_STATUS FIXED BIN(31) 5

DCL SY5$PUTMSG    ENTRY( ANY )?
DCL MSG_YEC(5)    FIXED BIN(31)

IF MOD(FMS-UM5-STATUS #2)=1 THEN RETURN ?
/* Save the FMS error code in message vector for PUTMSG */
/* Save the RMS status code in the message vector before
/* toakin further calls */
/* Detach the terminal to clean UP screen before printing
/* error message */
MSG_UEC(2) = FMS_UMS_STATUS ?
MSG_UEC(3) = 0    /* Required for non-system facility*/

MSG-UEC(5) = 0    /* In case RMS error «sfl needs it*/
CALLFMS = FDU$5TAT( FMSSTATUS » MSG_UEC <4> )
CALLFMS = FDU$DTERM( TCA )?
/* Set message vector count */
/* Output message(s) */
IF MSG_UEC <4> = 0
THEN MSG^UEC(1) = 1  /* No secondary status*/
ELSE MSG_UEC(l) = a
CALL SYS$PUTMSG( MSG_VEC )!
STOP
END CHKSTA
```

The subroutine could be simplified if the secondary call status were not needed. Instead of setting up a message vector, the RTL subroutine LIB$SIGNAL or LIB$STOP could be used. For example, the following could replace the entire body of the routine above:

```
DECLARE LIB$SIGNAL( FIXED BINARY(31) VALUE > i
CALL LIB$SIGNAL( FMS_VMS_STATUS )i
```

The Sample Application uses two subroutines to check for unexpected errors. The first few calls use a subroutine that calls STAT to obtain the status of the last Form Driver call. If the status obtained (which is the FMS, system independent status) is greater than zero, then the last call was successful and the subroutine returns. If the last call was not successful, the subroutine detaches the terminal (to clean up the screen) and prints the error codes reported.

After the first few calls, SAMP sets up two status recording variables, FMSSTATUS and RMSSTATUS. The Form Driver sets the FMS status in these variables for every Form Driver call thereafter.

The SAMPs use FMSSTATUS in two ways. They call another subroutine that checks for unexpected errors. This subroutine does not have to call STAT, but merely checks the FMSSTATUS variable. If there is an error, it performs the same error reporting as the first subroutine. The second way SAMP uses the FMSSTATUS is immediately after a call to PFT in the routine that simulates a GETAL call. Since the Form Driver set FMSSTATUS before returning to SAMP, SAMP can immediately refer to the value of FMSSTATUS.

The following is an adaptation (in BASIC) of the two SAMP routines described above. Consult the *VSI FMS Language Interface Manual* to see the routines used in several languages.

```
DEF FN.GETSTA
!*****************************************
! Subroutine GETSTA
! Check FMS status by calling STAT*
! If not success (>0)t print and stop
!_*****************************************

CALL FDU$STAT < FMSSTATUS* » RMSSTATUS* )
IF FMSSTATUS* > 0 THEN FNEXIT
CALL FDU$DTERM( TCA*< ) )
PRINT "FDV ERROR 11
PRINT " " *"FMS STATUS:" tFMSSTATUS*
PRINT " " *"RMS STATUS:" tRMSSTATUS*
STOP
FNEND
DEF FN •SRYCHK
! +*****************************************
! Subroutine SRUCHK
! Check FMS status bv looking at the status
! recording variables *
! –*****************************************
IF FMSSTATUS2 > 0 THEN FNEXIT
CALL FDU$DTERM ( TCA % < ) )
PRINT " FDU ERROR * "
PRINT t "FMS STATUS:" tFMSSTATUSZ

PRINT t "RMS STATUS : M > RMSSTATUSX
STOP
FNEND
```

3.11. Paging

Purpose

To facilitate collecting multiple pages of data on a single form.

Description

Consider the form in the Form Editor used to collect User Action Routine names and their associated data. A typical 23-line form might have space for five such pairs of data entries. To enter more than five UAR specifications, some extension technique must be employed. While scrolling is an excellent technique for accessing more data than will fit on a single screen, it is best used for accessing data records that can be compressed to a single line, since scrolled areas must be scrolled one line at a time.

The idea behind paging is to have the user enter an entire page of data without changing the screen, and then clear the fields to enter a new page of data. As the user tabs out of the last data field on the form, the form automatically advances to the next page. Similarly, if the user backspaces out of the first data field on the form, the form automatically moves back to the previous page (if there is one).

Since TAB and BACKSPACE are the default FMS keys used to move to the next or previous field in a form, no special action is required on the part of the operator to move among fields on different pages. Ideally, the operator should be able to move freely between pages to correct any previous entry just as he would modify a previous entry in a form. To make it clear what data is actually being edited, a page number could be displayed at the top of the form, or each data record could be preceded by a number that is updated to reflect the current page.

Programming Technique

Several FMS features are used to produce the paging behavior described above. The Form Driver PFT call (Process Field Terminator) is used to simulate GETAL processing. In addition, two special No Echo fields are added to the form one before the first data field, and the other after the last data field.

By having the application program know the names of these two fields, it is possible for the program to detect when the user moves out of the last data field of a form by requesting the name of the current field. Whenever the cur- rent field name matches the special first field, the program pages backward, similarly when the current field name matches the special last field, the program pages forward.

Alternatively, you can avoid the two special fields and the GETAL simulation by using the ILTRM (1) call with a GETAL call to have the Form Driver return illegal terminators. Your program can then identify the illegal terminator FDV$K_FT_ILG_NXT as a request to go to the next page, since the operator is attempting to go to the next field where there is none. A similar action can take place for FDV$K_FT_JLG_PRV. The program would have to filter out other illegal terminators.

Example

This technique is used in the Form Editor to enter Named Data and collect field completion UAR data. A subroutine in the Form Editor is used to simulate GETAL processing except that it returns a special terminator code whenever it encounters fields named FIRST and LAST.

3.12. FMS Advanced Programming

The following chapter discusses advanced programming techniques in two areas of FMS usage. The first section discusses how to enhance FMS performance; the second looks at the most effective way to design FMS overlaying forms. A working familiarity with these techniques will help FMS users to get the most out of FMS.

3.12.1. FMS Performance

This section provides information on how to maximize system performance while using VAX FMS. Subsections include FMS Library Performance and Form Driver Ordering of Calls.

3.12.1.1. FMS Library Performance

The maimer in which forms are stored and arranged in FMS libraries can either help or hurt performance. By observing the following, you are assured that your FMS form libraries will not hinder overall system efficiency:

Compression	always compress form libraries when transferring an application from the development cycle to the production cycle. Use the command: `* FMS /LIBRARY /CREATE library library .FLB`
Form Order	always insert commonly accessed forms in the library first.
Access Method	use the access method best suited to the application's forms, choosing from the following: 1. Memory Resident:

	• for forms that are displayed repeatedly such as main menus, • for times when the number of forms used by the application is small, or when the application is self contained. 2. Dynamic Memory Resident: • the dynamic memory resident list is searched first when looking for a form, • used for forms which are displayed for a selected function, such as sub-menus and “pages.” • also used when a memory resident form needs to be replaced at run-time with a form of the same name from a form library. 3. Media Resident: • for forms that are seldom accessed, i.e. Help forms, • used during application development when forms are modified frequently and relinking is not desired.

3.12.1.2. Form Driver Performance

Form Driver performance can be enhanced by using the most current calls available and by correctly ordering the Form Driver calls. FMS V2 calls are more efficient in both CPU time and I/O. Using FDV $INIT causes extra I/O on every Form Driver I/O call. Following these suggestions, and observing the correct ordering of FDV calls, as shown below, will maximize Form Driver performance.

Ordering of Form Driver Calls

Inefficient	**Efficient**
Displaying A Form	
FDV$LOAD	FDV$LOAD
FDV$DISP	FDV$DISPW
Initializing Fields	
FDV$DISP	FDV$LOAD
FDV$PUT	FDV$PUTAL
FDV$PUT	FDV$DISPW
FDV$PUT	
Using Dynamic Memory Resident Forms	
FDV$LOAD	FDV$READ
FDV$READ	FDV$LOAD
FDV$DISPW	FDV$DISPW
Checking Status	
FDV$xxxxx	FDV$SSRV

FDV$STAT	FDV$xxxxx
FDV$xxxxx	FDV$xxxxx
FDV$STAT	FDV$xxxxx
FDV$xxxxx	
FDV$STAT	

3.12.2. Designing Overlaying Forms

This section exists to help the programmer effectively use overlaying forms. To design overlaying forms, the programmer should have a clear understanding of FDV screen management. Therefore, the following sections include a list of FDV screen management rules, and an example of overlaying form design.

3.12.2.1. FDV Screen Management Rules

This section lists screen management rules. It consists of lists detailing when the FDV repairs screens, when the screen or workspace is broken, and the workspace repair sequence.

3.12.2.1.1. Screen repair occurs when:

- Operator requests a screen refresh
- Program calls FDV$RFRSH
- Returning from HELP
- Displaying a form
- Returning from a User Action Routine (UAR)
- Program calls FDV$PUTxx or FDV$GETxx to fields where the workspace that contained the fields was “broken.”

3.12.2.1.2. The screen or workspace is ’’broken" when:

- The program calls FDV$GETDL on a line other than the last
- The program calls FDV$PUTL on a line other than the last
- The area to clear of an overlaying form clears lines of the form
- The program calls FDV$CLEAR

3.12.2.1.3. Workspace Repair Sequence — The workspace is repaired in this order:

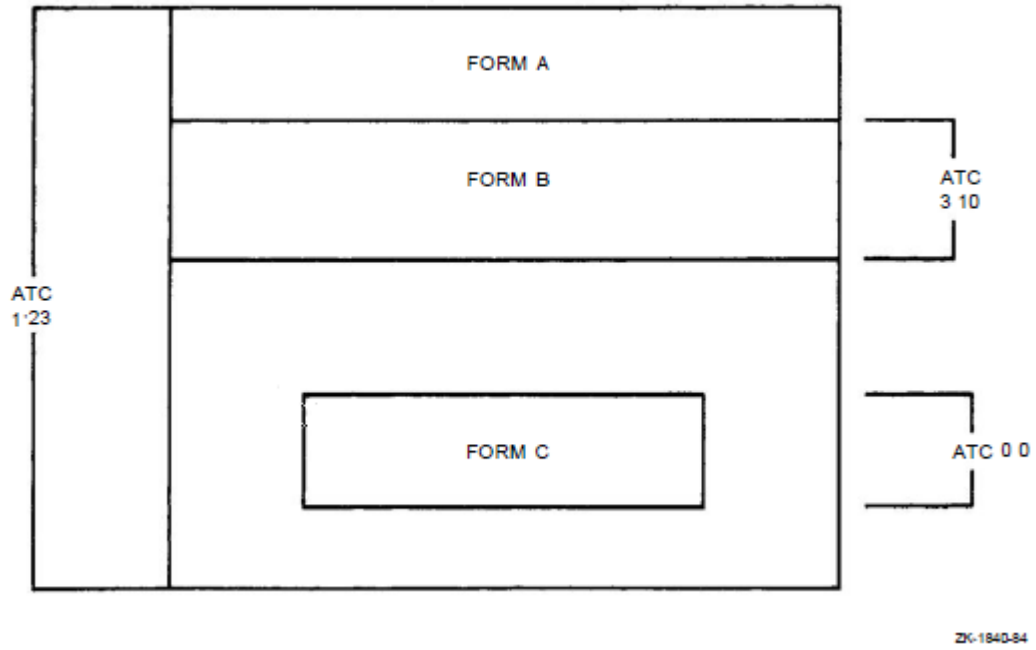
1. All workspaces, marked as displayed, are redisplayed in the order that they were attached, except for the current workspace.
2. The current workspace is redisplayed.

3.12.2.2. Overlaying Form Design

The following example includes three forms, each with its own values for Area To Clear (ATC). The ATC is the area to be cleared when a form is displayed. A call to FDV$CDISP clears the entire screen,

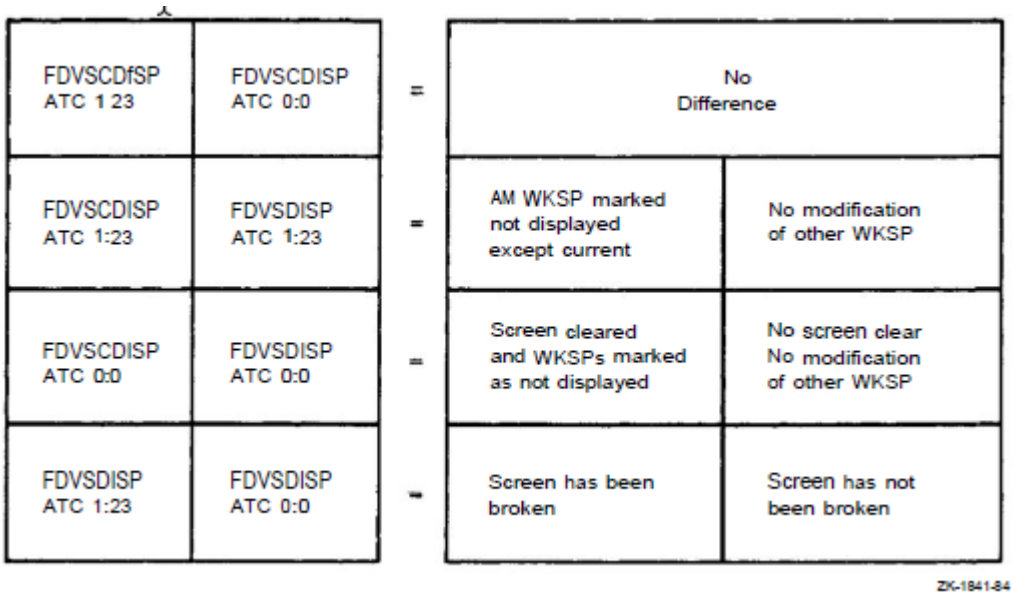
marks all workspaces as “not displayed” and displays the new form. However, to overlay forms as shown below, the programmer uses FDV$DISP, which displays the new form, clearing only those lines specified by the ATC.

Figure 3.1. Overlaying Forms



The following table compares different form displaying calls and area to clear combinations pertaining to overlaying forms, and their results.

Figure 3.2. Comparison of Overlaying Calls



Chapter 4. Linking the Application and Setting up the Terminals

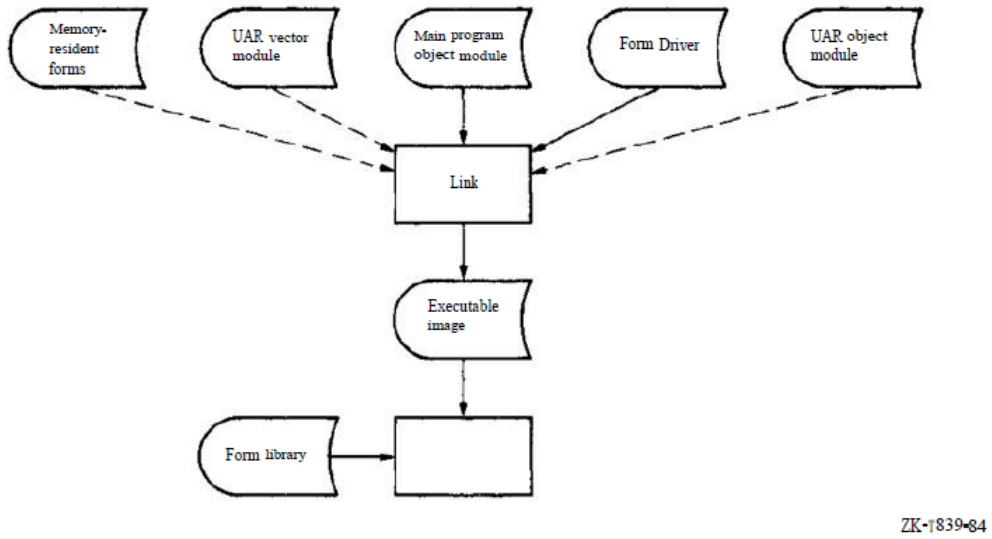
In the development of an FMS application, there are initially three processes:

- Creating forms and form libraries
- Writing the application program
- Writing user action routines (UARs)

These must, of course, be made into linkable object files. You then can link your object program with the Form Driver and, optionally, with any memory resident forms or UARs you wish to include. (See Figure 4.1.)

The Form Driver supports several terminal types and performs screen management correctly if you and your program follow certain procedures regarding setting the terminal characteristics and sending output to the terminal.

Figure 4.1. Linking the FMS Application



4.1. Linking

4.1.1. Linking with the Form Driver Library

Linking with the Form Driver Library is automatic, since the Form Driver routines are linked as a shared image:

```
* LINK MYPROG
```

4.1.2. Linking with Memory-Resident Forms

Using the Form Application Aid FMS/OBJECT gives you a linkable file of memory-resident forms:

```
* FMS /OBJECT form1ist … / OUTPUT=MRF .OBJ
```

Then:

```
$ LINK MYPROG i MRF
```

The advantages of keeping forms in memory are:

1. Faster access
2. Reduced disk overhead

The disadvantages are:

1. Larger executable disk images
2. Necessity to relink whenever a form changes

4.1.3. Linking with a UAR Vector

You use the Form Application Aid FMS/VECTOR to get a vector module. The module contains a table of UARs. You must link the modules with your program so the Form Driver knows where the UARs are when they are needed:

```
$ FMS/VECTOR FRMFIL 1.FRM » MYLIB.FLB/FORM_NAME = <FORM1 iFORM3)/OUTPUT =
 UARS1.OBJ
$ LINK MYRROG IUARSI application:
```

4.2. Terminal Use in FMS Programs

4.2.1. Terminal Characteristics

To support a variety of terminals, to allow type ahead, and to conform with VSI's long-term terminal software strategy, the Form Driver queries the operating system and not the terminal to find the terminal options and cur rent characteristics. This means that you should set VMS's knowledge of your terminal carefully before starting an FMS application program. One method of doing this is by issuing the following VMS command before running your application:

```
$ SET TERMINAL/INQUIRE
```

VMS will then query the terminal and record the terminal's characteristics correctly.

Between the issuing of this DCL command and its completion, you should not type ahead. Putting the SET TERMINAL/INQUIRE command in your login command file is a good idea.

If your terminal differs from VMS's knowledge of your terminal, your FMS application may not perform correctly. You can see what VMS thinks your terminal type is by issuing the following VMS command:

```
* SHOW TERMINAL
```

You will see displayed values for terminal attributes: type, width, advanced video, ANSI CRT, and DEC-CRT, among others. _Your terminal should be either a VT52, or it should have at least the ANSL CRT attribute. It should also have the DEC-CRT attribute as appropriate. See the VAX documentation for an explanation of these attributes. Make sure that the terminal width and the advanced video option attributes are set correctly for your terminal.

4.2.2. Direct Terminal Output

To optimize output to the terminal, the FMS V2 Form Driver assumes that it has sole control of the terminal once the terminal has been attached. Your program should not normally send output directly to an attached terminal; all output to an FMS terminal should go directly through FMS. Direct output to the terminal will likely be displayed at unexpected positions with unwanted video attributes, line attributes, or character set.

If your program changes the cursor position, video attributes, line attributes, or character set, subsequent calls on the Form Driver may yield incorrect results on the screen. The only circumstances under which you can send output directly to the terminal and not confuse the Form Driver is if you restore any changes you have made before calling the Form Driver again. The CLEAR_VA, FIX_SCREEN, and SCR_WIDTH calls may be used for this purpose.

4.2.3. Terminal State at Program End

Because the Form Driver minimizes output, it will leave the terminal in an awkward state if you fail to detach the terminal. Detaching the terminal clears video attributes, clears the last line, and positions the cursor at the bottom left comer of the screen with the terminal width set to that of the last form displayed.

Note that the design of some terminals makes it impossible to determine some of the attributes from the terminal so that neither the Form Driver nor your program can restore them once they have been changed:

- LEDs
- Character set
- Screen background

All these attributes remain in the state last set by the application program's form use and terminal control calls.

4.2.4. Firmware Bug Workaround

Many VT100s have a firmware bug that the Form Driver must work around.

(This workaround itself may appear to be a bug, but it is not.) The firmware bug appears when a scrolled area is immediately below a line having the double-high or double-wide attribute, and the terminal is in jump scroll mode. The bug is that the scrolled lines lose characters, or the terminal may enter self-test mode.

The workaround used by the Form Driver is to set the line above the scrolled area to be normal size during the scrolling and then reset it to double high or double wide afterward. This may be visually disturbing but does not affect your program. The visual effect can be avoided by not setting the line above a scrolled area to double high or double wide in the form definition.

Chapter 5. Form Driver Calls

The following sections contain descriptions of all Form Driver calls.

The call format shows the generic form of each call, with the FDV$ prefix you must specify with each call name (for example, FDV$ATERM - not just ATERM), and any arguments you must (or can) supply. Each argument is defined, and it is noted whether the Form Driver reads the argument from your program, writes (returns) it to your program, or both reads and writes (modifies) it. In your program you terminate a call by pressing the RETURN key unless the manual specifies otherwise.

You always specify an address for an argument rather than the actual data to be processed by the call or returned to your program. Each argument definition indicates the method of passing the argument:

- By reference – The address contains the value itself (used for passing integers).
- By descriptor – The address contains the address of a descriptor containing necessary information (used for passing character strings or integer arrays).

See the *VSI FMS Language Interface Manual* for details of the calling requirements unique to each language (for example, CALL FDV$ATERM). See also the appendix to this manual for a complete list of Form Driver calls showing the procedure parameter notation.

Description

Describes what the call does, any relationships to other calls, and restrictions on the use of the call.

Status Codes

Report for each call successful execution of the call or any failures that occur during processing of the call.

5.1. Alter Data Line Video Attributes

FDV$ADLVA (video)

video	The video attributes code. Set to 1, any or all of bits 0, 1, 2, and 3 specify any or all of the Bold, Blink, Reverse, and Underline attributes, respectively (decimal value in the range 0 to 15). If the value of this argument is negative, the video attributes are restored to their initial states. (Modified. Passed by reference.)

Attributes	***Value***
None	0
Bold	1
Blink	2
Blink and Bold	3
Reverse	4

Attributes	*Value*
Reverse and Bold	5
Reverse and Blink	6
Reverse, Blink, and Bold	7
Underline	8
Underline and Bold	9
Underline and Blink	10
Underline, Blink, and Bold	11
Underline and Reverse	12
Underline, Reverse, and Bold	13
Underline, Reverse, and Blink	14
Underline, Reverse, Blink, and Bold	15
Restore attributes to initial state (*-n* is any negative integer)	*-n*

Description

Lets you alter the video attributes for the current terminal's data line. You can also specify that these video attributes be restored to their original states.

The data line video attributes affect the appearance of (1) Form Driver messages, which are output to the bottom line of the screen, and (2) other lines of text, which are displayed by the execution of PUTL or GETDL calls.

This call returns the previous contents of video. The data line video attributes are returned encoded in the same format you used for the input of the new video attributes.

Status Codes

FDV$_ARG	Incorrect number of arguments.
FDV$_CAN	Call was terminated by a CANCL call.
FDV$_SUC	Successful completion of the call.
FDV$_SYS	Form Driver encountered system error response.
FDV$_TCA	No terminal control area (TCA) is defined.

5.2. Alter Field Context

FDV$AFCX (insovr,curpos[,fldnam[[,fldidx]])

insovr	The code indicating whether Insert or Overstrike mode is in effect for a field. (Read. Passed by reference.) 0 = No change 1 = Insert mode 2 = Overstrike mode

curpos	The cursor position within a field. The cursor position is 1 for the leftmost data character in the field, 2 for the next data character to the right, *n* for the rightmost character in the field, and *n* + 1 for the character position to the immediate right of the rightmost data character (the hanging cursor position). Field marker characters are not counted by the cursor. The range of the cursor, 1 to *n* + 1, is limited to the number of data characters in the field plus 1. (Read Pass^ by reference.) For fixed-decimal fields, the range of the cursor is 1 to + 2, because the decimal point is counted even though it is not a data character. This allows the cursor to be positioned on the decimal point, in the hanging cursor position for the left-hand part of the field.
fldnam	The field name. If **fldnam** is not specified, the current field is assumed. (Read. Passed by descriptor.)
fldidx	The field index. (Read. Passed by reference.)

Description

Alters the default input mode of a field. This call specifies both the Insert/Overstrike mode of the field and the cursor position in the field for any GET-type call operation affecting the field.

The new context specified by this call remains in effect until the field is processed as part of any GET-type call. The context is restored to its default state when the operator exits the field. Note that if the input processing is part of a GETAL or GETAF call, passing through the field in the normal course of moving the cursor about on the screen restores the default field context.

Status Codes

FDV$_ARG	Incorrect number of arguments.
FDV$_CAN	Call was terminated by a CANCL call.
FDV$_DSP	Form contains only display-only fields.
FDV$_FLD	Field does not exist, or index value is invalid for field.
FDV$_INI	No workspace is defined.
FDV$_NFL	No form loaded in current workspace.
FDV$_NOF	Form contains no fields.
FDV$_SUC	Successful completion of the call.
FDV$_TCA	No terminal control area (TCA) is defined.
FDV$_VAL	The value of **insovr** or **curpos** is outside the allowed range.

5.3. Alter Field Video Attributes

FDV$AFVA ([video!,fldnam[,fldidx]])

video	The video attributes code. Set to 1, any or all of bits 0, 1, 2, and 3 specify any or all of the Bold, Blink, Reverse, and Underline attributes, respectively (decimal value in the range 0 to 15).If the value of this argument is negative, the video attributes are restored to their initial states. (Modified. Passed by reference.)

Attributes	*Value*
None	0
Bold	1
Blink	2
Blink and Bold	3
Reverse	4
Reverse and Bold	5
Reverse and Blink	6
Reverse, Blink, and Bold	7
Underline	8
Underline and Bold	9
Underline and Blink	10
Underline, Blink, and Bold	11
Underline and Reverse	12
Underline, Reverse, and Bold	13
Underline, Reverse, and Blink	14
Underline, Reverse, Blink, and Bold	15
Restore attributes to initial state (*-n* is any negative integer)	*-n*

fldnam	The field name. If **fldnam** is not specified, the current field is assumed. (Read. Passed by descriptor.)
fldidx	The field index. (Read. Passed by reference.)

Description

Lets you alter the video attributes of a field in a form. You can also use this call to restore these video attributes to their original states. The field video attributes immediately change on the screen and remain in effect until you either issue another AFVA call to change them, or you redisplay the form by issuing a DISP or CDISP call.

This call returns the previous contents of video. The video attributes for a field are returned encoded in the same format you used for the input of the new video attributes.

If you alter the video attributes of a field by issuing an AFVA call, you cancel any input highlighting for the field. That is, if highlighting is in effect for the form, it will not be used for this field. You restore highlighting by restoring the default video attributes of the field.

Status Codes

FDV$_ARG	Incorrect number of arguments.
FDV$_CAN	Call was terminated by a CANCL call.
FDV$_FLD	Field does not exist, or index value is invalid for field.
FDV$_INI	No workspace is defined.
FDV$_NFL	No form loaded in workspace.
FDV$_NOF	Form contains no fields.

FDV$_SUC	Successful completion of the call.
FDV$_SYS	Form Driver encountered system error response.
FDV$_TCA	No terminal control area (TCA) is defined.

5.4. Attach Terminal

FDV$ATERM (tea,size,channel[,trmnal[,faketrmtyp[,options)]]]

<table>
<tr><td>tea</td><td colspan="2">The name of a terminal control area. The TCA size, contained in the descriptor, must be at least 12 bytes. (Modified. Passed by descriptor.)</td></tr>
<tr><td>size</td><td colspan="2">The size of the TCA in bytes. (Read. Passed by reference.) (Ignored in VMS.)</td></tr>
<tr><td>channel</td><td colspan="2">The logical I/O channel number for the terminal. (Read. Passed by reference.)</td></tr>
<tr><td>trmnal</td><td colspan="2">The name of the terminal to be associated with the TCA. If trmnal is omitted, the default is SYS$INPUT. (Read. Passed by descriptor.)</td></tr>
<tr><td>faketrmtyp</td><td colspan="2">The name of the terminal type the Form Driver assumes for batch use. The only valid string supported currently is “VT100”. If this argument is given, there is no real input or output to the terminal.(Read. Passed by descriptor.)</td></tr>
<tr><td rowspan="5">options</td><td colspan="2">An integer specifying which Form Driver options are to be associated with this terminal. When specifying the option, the integer is a longword with the following bits:</td></tr>
<tr><td>BIT Setting</td><td>Meaning</td></tr>
<tr><td>0</td><td>The screen shall not be cleared when attaching the terminal.</td></tr>
<tr><td>1</td><td>The Form Driver clears the terminal’s video attributes after each call (see CLEARJVA)</td></tr>
<tr><td>2</td><td>AST support is not required. Requesting this option improves performance (see AST considerations).</td></tr>
<tr><td>3-31</td><td>Reserved by Digital.</td></tr>
</table>

Description

Attaches a terminal to the Form Driver. You must issue an ATERM call for any terminal the Form Driver needs to access for form processing. This call makes the attached terminal the current terminal.

When you specify a channel, you are specifying a logical channel. FMS asks VMS to assign a physical channel. If you want the Form Driver to use a particular channel, first issue ATERM and then issue TCHAN. Form Driver logical channel numbers are all interpreted modulus 256.

The channels specified in the Form Driver calls ATERM, LCHAN, and LOPEN are strictly local to FMS and have no relationship to Logical Unit Numbers used by FORTRAN and BASIC. These channel numbers provide a means of reference only. The Form Driver keeps an association list of all logical channels currently in use by the application program. Logical terminal numbers and logical form library numbers must not conflict; that is, a logical terminal channel number cannot be used as a logical form library channel number.

ATERM lets you specify a terminal control area for this terminal. This area maintains all necessary information about current terminal characteristics and associations. Other calls refer to these areas implicitly or explicitly whenever they need to indicate a particular terminal. The terminal must be VT200-, VT100 or VT52-compatible.

When the faktrmtyp argument is used in ATERM to attach a fake terminal, the terminal so attached is defined as a VT100 type terminal with 65 lines and 132 columns (instead of just 24 lines). This gives applications the opportunity to use such a fake terminal to produce a line printer report. Calls to RETFL, for such a terminal, can access lines 1 through 65. Since the Form Language and the Form Editor allow the production of forms only 23 lines long, three forms in three workspaces (with appropriate offsets) are necessary to produce a full screen of output.

If the faketrmtyp argument is not supplied (or is null), the terminal specified in the trmnal argument is attached in the normal fashion. When faketrmtyp is specified as the descriptor of a character string containing “VT100”, no actual terminal attachment is made, and the trmnal argument is ignored.

Any calls for input on the terminal specified by this TCA result in FDV$_ITT errors. All output normally directed to the screen is suppressed. However, calls that normally produce output, such as CDISP and PUT, still modify the workspace. The Form Driver can then be used as an output formatting tool.

The faketrmtyp argument causes subsequent calls to RETFL to produce line images of what would have been displayed.

The options argument is useful when you want your program to retain part of the current screen while FMS is using another part of the screen. The options argument is further defined by setting the two low order bits of the argument.

Bit 0 is described above. Bit 1 set to 1 directs the Form Driver to reset the video attributes and character set of this terminal to the clear state (no video attributes and character set shifted in) after every Form Driver call. Bit 1 set to 0 directs the Form Driver to leave the attributes of the terminal set between calls to achieve minimal output.

The use of the second bit in ATERM makes it easier to perform direct screen management between Form Driver calls, at the price of additional output on every Form Driver call that touches the screen. A program that performs direct screen management can choose either to attach the terminal with this bit set, or to_ call FDV$CLEAR_VA at appropriate times. Using FDV$CLEAR_VA requires more care on the part of the program, but allows the Form Driver to optimize output slightly.

AST support requires a great deal of overhead in the Form Driver. If your application does not need AST reentrant support, setting bit 2 greatly reduces the overhead in the Form Driver and will improve its performance. However, you must not set this bit if your applications use the Form Driver in an AST fashion. This will cause unexpected results.

In addition, ATERM clems the screen and turns the LEDs off.

If you do not specify a terminal, the default terminal is attached.

Status Codes

FDV$_ARG	Incorrect number of arguments.
FDV$_CAN	Call was terminated by a CANCL call.
FDV$_ICH	Logical channel specified was either in use or invalid.
FDV$_ITT	Illegal terminal type.
FDV$_JVM	Not enough virtual memory could be allocated for the TCA.
FDV$_STA	Size of specified TCA is too small.
FDV$_SUC	Successful completion of the call.

FDV$_SYS	Form Driver encountered system error response.
FDV$_VAL	Either the faketrmtyp argument was given and it was not “VT100”, or the options argument was given and it was not within the range of 0-7.

5.5. Attach Form Workspace

FDV$AWKSP (wksp,size)

wksp	The form workspace location. The length (in bytes) recorded in the descriptor, must be a value of at least 12. (Modified. Passed by descriptor.)
size	An estimate of the workspace size in bytes. If the size turns out to be too small, the Form Driver automatically increases it. (Read. Passed by reference.)

Description

Attaches a form workspace to the current terminal. You must issue an AWKSP call for any form your application processes on any terminal. This call makes the attached workspace the current workspace.

The Form Driver uses the 12-byte area specified in the wksp descriptor as linkage to an area that it allocates in virtual memory. This area is used to store the variable part of a form description.

The size argument is an estimate of the storage space needed for a loaded form. If you underestimate the amount of space you need, the Form Driver automatically allocates more space — but a large enough estimate can save time.

You can use the FMS/DIRECTORY command to find out the workspace size you need for each form you expect to use. (See the *VSI FMS Utilities Reference Manual*.)

Status Codes

FDV$_ARG	Incorrect number of arguments.
FDV$_CAN	Call was terminated by a CANCL call.
FDV$_IMP	Length specified in wksp descriptor is not large enough.
FDV$JTVM	Not enough virtual memory could be allocated for the workspace.
FDV$_SUC	Successful completion of the call.
FDV$_TCA	No terminal control area (TCA) is defined.

5.6. Ring Terminal Bell

FDV$BELL

Description

Rings the terminal bell. If the current terminal is defined, this call rings its bell. If the current terminal is not defined, the call rings the bell on the application program’s default terminal. This call rings the bell regardless of the signal mode (see the SSIGQ and SIGOP call descriptions).

Status Codes

FDV$_ARG	Incorrect number of arguments.
FDV$_CAN	Call was terminated by a CANCL call.
FDV$_SUC	Successful completion of the call.
FDV$_SYS	Form Driver encountered system error response.

5.7. Cancel Call

FDV$CANCL

Description

Causes any other Form Driver call presently being processed on the current terminal to be terminated with the error condition FDV$_CAN.

This call has no effect unless it is executed from an AST service routine or a UAR, since no other call can otherwise be executing.

When executed, this call causes two things to happen:

1. All I/O operations associated with the current terminal are canceled.

2. Until the cancellation processing is complete, any other call involving the same TCA as that of the call being canceled is itself canceled when issued. This action is taken to ensure that all calls issued by a UAR are canceled as well.

As a result of these two activities, the call normally terminates almost immediately. If a call has called a UAR, however, the call-processing code cannot terminate processing until the UAR returns control to the Form Driver. Upon return from a UAR, the Form Driver checks to see if the TCA is marked as processing a CANCL operation and if it is, the call is terminated.

If a call is canceled after its UAR has begun executing, any subsequent calls the UAR might issue will also be terminated with the status of FDV$_CAN.

CANCL returns FDV$_SUC whether or not any call is canceled.

Status Codes

FDV$^ARG	Incorrect number of arguments.
FDV$_SUC	Successful completion of the call.
FDV$_SYS	Form Driver encountered system error response.
FDV$_TCA	No terminal control area (TCA) is defined.

5.8. Clear Screen and Display Form

FDV$CDISP (frmnamf [,offset])

frmnam	The name of the form.{Read. Passed by descriptor.)

offset	The position of the form on the screen. (Read. Passed by reference.) • If **offset** contains**0**, the Form Driver positions the form on the screen as specified in the form description. • **If offset** contains a nonzero value, the Form Driver moves the form up (if the value is negative) or down (if the value is positive) by the amount specified.

Description

Executes a CLEAR call and then a DISP call, clearing all forms from the screen (marking them as undisplayed) and displaying a new form. If any other workspaces are attached to the current terminal, your program can redisplay their forms by issuing DISPW calls on their workspaces. See the description of the DISP call for additional details.

Status Codes

FDV$_ARG	Incorrect number of arguments.
FDV$_CAN	Call was terminated by a CANCL call.
FDV$_FCH	Form was not memory resident, and when the Form Driver attempted to search for it in a form library, the current library channel was not open.
FDV$_FNM	Binary form description could not be found either in the form library or in the list of memory-resident forms.
FDV$_FRM	Form description is invalid.
FDV$_IFU	Workspace cannot be loaded at this time because it is the workspace for a currently active UAR.
FDV$_INI	No workspace is defined.
FDV$_IOR	I/O error occurred while Form Driver was reading in the form from the form library. The I/O error code is recorded in the current state. You can obtain it by issuing the STAT call.*
FDV$_JVM	Not enough virtual memory could be allocated for the workspace.
FDV$_LIN	Starting offset is invalid. Form does not fit on the screen if offset by the amount specified.
FDV$_SUC	Successful completion of the call.
FDV$_SYS	No terminal control area (TCA) is defined.
FDV$_TCA	Form Driver encountered system error response.

5.9. Clear Screen

FDV$CLEAR ([lme.rLr[,linecnt.rl.r]])

line	The number of the first line of the screen to be cleared. A value of zero specifies the top of the screen. (Read. Passed by reference.)
linecnt	The number of lines to clear. If you specify 0, all lines from the line you specified in the line argument to the bottom of the screen are cleared. (Read. Passed by reference.)

Description

Clears all or part of the screen. If **line** is greater than zero, the screen is cleared from that line down; otherwise, the screen is cleared from the top down. If **linecnt** is greater than zero, it specifies the number of lines to be cleared. If it is zero, the screen is cleared to the bottom of the screen.

A refresh operation (whether by your program or by the operator) following a CLEAR operation restores the entire screen, including the cleared area.

Following the CLEAR operation, the cursor is positioned at the left most character position on the first line cleared.

Status Codes

FDV$_ARG	Incorrect number of arguments.
FDV$_CAN	Call was terminated by a CANCL call.
FDV$_LIN	Call specifies that some line not on the screen be cleared.
FDV$_SUC	Successful completion of the call.
FDV$_SYS	Form Driver encountered system error response.
FDV$_TCA	No terminal control area (TCA) is defined.

5.10. Clear Video Attributes

FDV$CLEAR_VA

Description

Clears the screen of video attributes and sets certain other terminal attributes. It is useful when you want your program to write to, or control the screen, directly, while a terminal is still attached to FMS. CLEAR forms the following operations:

For VT52-type terminals:

- Shifts in character set.

For VT100-type terminals:

- Shifts in character set.
- Turns off the screen's video attributes.
- Sets newline mode.
- Resets the scrolling area.
- Sets absolute origin mode.

This call does not affect the character sets currently selected in G0 or G1 for VT100-compatible terminals, the width of the screen, or the background color of the screen.

CLEAR-VA should be issued in the following situations:

- Just before your program starts its own direct screen management after FMS has been controlling the screen. This clears the screen of any attributes (most importantly, video attributes) that FMS may have left, so that your program can start with a known, clean screen.
- Just before your program calls FMS after your program has changed any of the above attributes of the screen. This clears the screen of any attributes your program may have left, so that FMS can continue with a known screen.

See also the description of SCR_WIDTH for information on a call that informs the Form Driver that your program has changed the width of the screen.

Status Codes

FDV$_ARG	Incorrect number of arguments.
FDV$_CAN	Call was terminated by CANCL call.
FDV$_SUC	Success.
FDV$_TCA	No current terminal.

5.11. Remove Form from Memory-Resident Form List

FDV$DEL(frmnam)

frmnam	The name of the form. (Read. Passed by descriptor.)

Description

Deletes a memory-resident form from the list of memory-resident forms you loaded with the READ call. You cannot delete with this call those memory resident forms that are built into the application program.

If the form you specified is not found, or if it is in the set of forms built into the application, this call returns the status code of FDV$_FNM.

Status Codes

FDV$_ARG	Incorrect number of arguments.
FDV$_CAN	Call was terminated by a CANCL call.
FDV$_FNM	Binary form description could not be found in the set of memory-resident forms that could be deleted (that is, those loaded by a READ call).
FDV$_SUC	Successful completion of the call.

5.12. Define Keyboard

FDV$DFKBD (defkbd,kbdnum)

defkbd	An array of key functions and key codes. (Read. Passed by descriptor.)

kbdmim	The number of entries in the defkbd array. Each entry is a pair of array slots. Thus, the length of the array must be at least two times kbdnum. (Read. Passed by reference.)

Description

The Form Driver has 17 functions and provides default keys to perform these functions. However, the user can override these default Form Driver function key assignments using defkbd. The defkbd argument is a one-dimensional array of words, with kbdnum pairs of entries. (That is, defkbd is expected to be two times kbdnum words.) The first word of each pair is a Form Driver key function, as defined below. The second word is a Form Driver key code as defined in Chapter 2. Two special key codes are FDV$K_KF_DFLT and FDV$K_KF_NONE.

1. FDV$K_KF_DFLT declares that the key code for the key function is to be the default. (This restores the default after a previous change.)

2. FDV$K_KF_NONE declares that no key code is to be associated with the key function. (The key function is deleted.)

The key codes in defkbd replace the current key codes for the current TCA. Note that not all key functions need be specified in a given call. Only those in defkbd are affected. The defkbd array is merged with the currently active table to produce the table used by the Form Driver.

As a result of the merging of defkbd and the current definitions, no key code can be assigned more than one key function. Assigning FDV$K_KF_DFLT to a key function assigns all key code defaults to that key function.

The following table lists the FDV functions and the default key assignments.

Function Name	Description	VT100 Key Sequence	DFKBD Value
FDV$K_KF_DLCHR	Delete character	DELETE	1
FDV$K_KF_CRSRT	Move cursor right	Rightarrow	2
FDV$K_KF_CRSLF	Move cursor left	Leftarrow	3
FDV$K_KF_DLFLD	Delete field	LINEFEED	4
FDV$K_KF_INS	Set Insert mode	PF1 PF3	5
FDV$K_KF_OVR	Set Overstrike mode	PF3	6
FDV$K_KF_GOLD	Gold sequence starter	PF1	7
FDV$K_KF_RESET	Reset Gold sequence	PF1 DELETE	8
FDV$K_KFJRFRSH	Refresh screen	CTRL/R	9
FDV$K_KF_JHELP	Help	PF2	10
FDV$K_KF_NXT	Next field	TAB	11
FDV$K_JCF_JPRV	Previous field	BACKSPACE	12
FDV$K_KF_NTR	Enter Form	RETURN or ENTER 13	13
FDV$K_KF_SBK	Scroll backward	Uparrow	14
FDV$K_KF_SFW	Scroll forward	Downarrow	15
FDV$K_KF_XBK	Exit scrolled area backward	PF1 Uparrow	16

FDV$K_KF_XFW	Exit scrolled area forward	PF1 Downarrow	17

The following example shows how to use the DFKBD call to switch the functions of the RETURN and TAB keys. After this call is executed, RETURN (and the ENTER key) will mean Next Field (FDV$K_FT_NXT), and TAB will mean Enter Form (FDV$K_FT_NTR). The example is given in FORTRAN.

```
INTEGER TCA ( 3 )
INTEGER _*2 KEYTABLE ( 4 ) / FDY$K_KF_NTR * 1033 *
1                            FDO$K_KF-NXT * 1037 /
CALL FDV$ATERM ( ZDE5CR ( TCA > 1 2 >1 )
CALL FD0$DFKBD ( ZDESCR ( KEYTABLE ) * 2 )
```

Status Codes

FDV$_ARG	Incorrect number of arguments.
FDV$_CAN	Call was terminated by a CANCL call.
FDV$_KEX	Too many key codes were defined for some key function.
FDV$_JQF	Illegal key function was given.
FDV$_KIL	Illegal key code was given; that is, the key was not on the list in Chapter 2.
FDV$_KTW	Key code was given two separate key functions.
FDV$_SUC	Successful completion of the call.
FDV$JTCA	No terminal control area (TCA) is defined.

5.13. Display Form

FDV$DISP (frmnam[,offset])

frmnam	The name of the form. (Read. Passed by descriptor)
offset	The position of the form on the screen. (Read. Passed by reference.) • If offset contains 0, the Form Driver positions the form on the screen as specified in the form description. • If offset contains a nonzero value, the Form Driver moves the form up (if the value is negative) or down (if the value is positive) by the amount specified.

Description

Displays a form, clearing the portion of the screen specified as the clear area in the form description. Any portion of the screen not cleared is overwritten by nonblank portions of the form. If offset has a value of zero, the form is positioned as specified in the form description. If offset contains a nonzero value, the Form Driver moves the form up (if the value is negative) or down (if the value is positive) by the amount specified.

If the form does not fit on the screen (that is, if some portion of background text falls outside the area of line 1 through line 23, or line 1 through line 14 for a non-AVO terminal in 132-column mode), the Form Driver returns the FDV$_LIN status code.

If the form specifies a screen width different from the current width, the formwide screen width attribute determines the Form Driver's action:

- If the form does not have the screen width attribute, the Form Driver does not modify the width. No error can occur with 80-column forms because they always fit. Forms having 132 columns do not fit if the screen is currently set for 80 columns wide.

- If the form does have the screen width attribute, the Form Driver always modifies the screen width. If the form is an 80-column form, but a 132-column form is already displayed on the screen, the 132-column form is removed from the screen and marked as undisplayed. The form specified in the call is displayed.

For terminals not capable of being switched to 132 columns, 132-column forms cause an error.

If the form being displayed specifies that the screen video be modified when the form is displayed, the screen video is set as directed, regardless of what screen video specifications are associated with any other form already displayed from other attached workspaces.

If the background text or fields of the form displayed overlap any background text or fields of another form already displayed on the screen, the previous text is replaced. If the screen is refreshed, however, the final screen image may be changed because the workspaces are redisplayed in the order they were attached. The workspace's form that is current at the time of the refresh is displayed properly.

The following may clarify the screen management role of the Form Driver when overlaid forms are present, and when your program issues PUTL and GETDL calls to reference lines that are parts of forms.

Whenever the Form Driver is directed to output a value to the screen by one of the PUT-type field calls (PUT, PUTAL, PUTD, PUTDA, or PUTSC), or is directed to request input from the operator by one of the GET-type field calls (GET, GETAF, GETAL, or GETSC), it first checks to see if the form containing the field is still intact on the screen. If the form has been disturbed in any way, the Form Driver redisplays it.

A form is disturbed in one of two ways:

1. Part of it has been overlaid by another form (in a subsequent DISPW call, RFRSH call or operation, or help request).

2. Part of it has been overlaid by a PUTL or GETDL call.

No matter what part of the form has been overlaid, the Form Driver ensures that the entire form is displayed.

The Screen Area to Clear attribute of a form is included in the description of the form, so that the form designer should consider how much of the screen should be included in this attribute at design time. Even if the form does not specify text or fields for a line, the Screen Area to Clear attribute may specify that the line be blank when the form is displayed.

The Form Driver honors the form description whenever the form is referenced. If you find a form being redisplayed when you do not expect it, it is most likely that part of the form has been overwritten.

Status Codes

FDV$_ARG	Incorrect number of arguments.

FDV$_CAN	Call was terminated by a CANCL call.
FDV$_FCH	Form was not memory resident, and when the Form Driver attempted to search for it in a form library, the current library channel was not open.
FDV$_FNM	Binary form description could not be found either in the form library or in the list of memory-resident forms.
FDV$_FRM	Form description is invalid.
FDV$_IFU	Workspace cannot be loaded at this time because it is the workspace for a currently active UAR.
FDV$_INI	No workspace is defined.
FDV$_IOR	I/O error occurred while Form Driver was reading in the form from the form library. The I/O error code is recorded in the current state. You can obtain it by issuing the STAT call.
FDV$_TVM	Not enough virtual memory could be allocated for the workspace.
FDV$_LIN	Starting offset is invalid. Form does not fit on the screen if offset by the amount specified.
FDV$_SUC	Successful completion of the call.
FDV$_SYS	Form Driver encountered system error response.
FDV$_TCA	No terminal control area (TCA) is defined.
FDV$_WID	The form to be displayed will not fit on the screen (132-column form on a VT52).

5.14. Display Loaded Form

FDV$DISPW ([offset])

offset	The position of the form on the screen. (Read. Passed by reference.) • If offset contains 0, the Form Driver positions the form on the screen as specified in the form description. • If offset contains a nonzero value, the Form Driver moves the form up (if the value is negative) or down (if the value is positive) by the amount specified.

Description

Displays on the current terminal a form already loaded in a workspace. A form can be resident in a workspace but undisplayed for any of the following reasons:

- It was previously loaded by a LOAD call.
- It was removed from the screen as a side effect of a CDISP call.
- It was a 132-column form that was removed from the screen as a side effect of another display call that loaded an 80-column form having the screen width attribute (see the description of the DISP call).
- It was marked as not displayed by the execution of an NDISP call.

DISPW clears any portion of the screen (possibly offset by the offset argument) that was specified as an area to be cleared in the form description. Any other portion of the screen is modified only if it is overwritten by the back- ground or field text of the form displayed. See the description of the DISP call for additional details.

Status Codes

FDV$_ARG	Incorrect number of arguments.
FDV$_CAN	Call was terminated by a CANCL call.
FDV$_FCH	Form was not memory resident, and when the Form Driver attempted to search for it in a form library, the current library channel was not open.*
FDV$_FNM	Binary form description could not be found either in the form library or in the list of memory-resident forms.
FDV$_LNI	No workspace is defined.
FDV$_LIN	Starting offset is invalid. Form does not fit on the screen if offset by the amount specified.
FDV$_SUC	Successful completion of the call.
FDV$_SYS	Form Driver encountered system error response.
FDV$_TCA	No terminal control area (TCA) is defined.
FDV$_WID	Form being displayed does not fit on the screen (132-column form on a VT52).

5.15. Define Comma as Decimal Point

FDV$DPCOM ([dpmode])

dpmode	A value determining what the decimal point character is to be: 1 = Comma is accepted exclusively as the decimal point. 0 = Period is restored to its role as the decimal point. (Read. Passed by reference.)

Description

Defines the comma, or redefines the period, exclusively, as the decimal point for fields containing the signed numeric (N) picture. The decimal point is returned to your program as part of the field value (unlike the decimal point in fixed-decimal fields). The default is the period.

Status Codes

FDV$_ARG	Incorrect number of arguments.
FDV$_CAN	Call was terminated by a CANCL call.
FDV$_SUC	Successful completion of the call.
FDV$_TCA	No terminal control area (TCA) is defined.
FDV$_VAL	The value of **dpmode** is outside the allowed range.

5.16. Detach Terminal

FDV$DTERM (tea)

tea	The name of a terminal control area. {Modified. Passed by descriptor.)

Description

Detaches a terminal from the Form Driver. All workspaces associated with the terminal are detached, and then the terminal itself is detached. No further Form Driver activity occurs with this terminal after DTERM is executed. Any forms displayed on the detached terminal remain on the screen.

When a terminal is detached, the character video attributes are cleared, the scroll area (VT100s only) is set to the full screen, the bottom line is cleared, and the cursor is placed at the leftmost position on the bottom line.

Normally, when the TCA is detached from the FMS application program, its associated terminal is detached from the program. An exception to this rule occurs if the channel was specified by the TCHAN call. Because TCHAN, alone, specifies a physical channel rather than a logical one, the terminal is not detached following a DTERM call.

Status Codes

FDV$_ARG	Incorrect number of arguments.
FDV$_CAN	Call was terminated by a CANCL call.
FDV$_FVM	An error occurred freeing virtual memory allocated to the application.
FDV$_CFU	Terminal cannot be detached at this time because it is the terminal for a currently active UAR.
FDV$_SUC	Successful completion of the call.
FDV$_SYS	Form Driver encountered system error response.
FDV$_TCA	No terminal control area (TCA) is defined.

5.17. Detach Form Workspace

FDV$DWKSP (wksp)

wksp	The form workspace. (Modified. Passed by descriptor.)

Description

Detaches a form workspace from the current terminal. Any form that is currently displayed remains on the screen.

Status Codes

FDV$_ARG	Incorrect number of arguments.

FDV$_CAN	Call was terminated by a CANCL call.
FDV$_FVM	An error occurred freeing virtual memory allocated to the application.
FDV$_IFU	Workspace cannot be detached at this time because it is the workspace for a currently active UAR.
FDV$_INI	Workspace does not exist or is not associated with the current terminal.
FDV$_SUC	Successful completion of the call.
FDV$_TCA	No terminal control area (TCA) is defined.

5.18. Repair Overwritten Lines of Terminal Screen

FDV$FIX_SCREEN

Description

Programs which perform direct screen management may overlay lines of forms displayed by FMS. Calling FIX-SCREEN will repair these lines with a minimum of output. FIX-SCREEN is similar to RFRSH, but with two exceptions: FIX SCREEN does not clear the screen first, and it outputs only those lines which it knows to have been cleared.

Whenever information is sent to a screen field (by a PUT-type call) or requested of the screen (by a GET-type call), the Form Driver checks the lines of the form. If any line has been cleared as described above, the Form Driver then repairs the affected lines of the screen by calling FIX SCREEN internally. Thus, your program need do nothing if the screen has been affected by calls on the Form Driver, such as CLEAR, PUTL, GETDL, DISPW, CDISP, or DISP, since the Form Driver knows and will fix the screen before the next I/O operation affecting fields.

However, if your program performs direct screen management (that is, it affects the screen without calling the Form Driver) the Form Driver will not know when the screen has been affected and will not automatically fix it. If you wish the Form Driver to restore the screen to the proper state after your own direct screen management, you must first clear the lines through the

Form Driver. One way of doing this is through the CLEAR call. After the CLEAR call the Form Driver will know that the lines need to be restored to their proper form state.

5.19. Get Value for Specified Field

FDV$GET (fldval,fldtrm41dnain[,fldidx])

fldval	The field value. The value consists of data characters, but no field-marker characters. If the operator does not enter a character for every position in the field, the Form Driver fills the empty positions with fill characters. (Written. Passed by descriptor.)
fldtrm	The field terminator that the operator entered to terminate input to the field. (Written. Passed by reference.)
fldnam	The field name. (Read. Passed by descriptor.)

fldidx	The field index. (Read. Passed by reference.)

Description

Waits for the operator to type a value into the field you specified, and records the field terminator in fldtrm. If fldnam starts with an asterisk (*), the Form Driver prompts the operator for input to the first modifiable field. If the field is in a scrolled area, the operation is performed in the current scrolled line for that area.

If the terminator is a function key not reserved for FMS, the form's function key UAR is called. The function key UAR may suppress the terminator, ignore it, or change it before subsequent processing occurs.

If the terminator is not a Previous Field terminator or a function key, the Form Driver checks the field for Response Required, Must Fill, and field completion UAR requirements. If any requirements are not fulfilled, the operator must continue input.

The value and its terminator are recorded in the workspace and are returned to your program. If a field value is changed by the operator when this call is executed, the status code returned is FDV$ MOD instead of FDV$ SUC.

Status Codes

FDV$_ARG	Incorrect number of arguments.
FDV$_CAN	Call was terminated by a CANCL call.
FDV$_DSP	Form contains only Display Only fields, or the specified field is Display Only.
FDV$_FLD	Field does not exist, or index value is invalid for field.
FDV$_INI	No workspace is defined.
FDV$_MOD	Field value in fldval has been modified by the operator. Otherwise, this code is the same as FDV$_SUC.
FDV$_NDS	Form is marked as being not displayed, so no input is possible.
FDV$_NFL	No form loaded in workspace.
FDV$_NOF	Form contains no fields.
FDV$_STR	Value being returned is too large for the variable allocated for it.
FDV$_SUC	Successful completion of the call.
FDV$_SYS	Form Driver encountered system error response.
FDV$_TCA	No terminal control area (TCA) is defined.
FDV$_TMO	Operator took longer to respond than allowed by the timeout value associated with the current terminal.
FDV$OJAR	UAR returned illegal code.
FDV$_UDP	UAR depth exceeded.
FDV$_UNF	UAR specified but not found.

5.20. Get Value for Any Field

FDV$GETAF (fldval,fldtrm,fldnam[,fldidx])

fldval	The field value. The value consists of data characters but no field-marker characters. If the operator does not enter a character for every position in the field, the Form Driver fills the empty positions with fill characters. (Written. Passed by descriptor.)
fldtrm	The field terminator that the operator entered to terminate input to the field. (Written. Passed by reference.)
fldnam	The field name. (Written. Passed by descriptor.)
fldidx	The field index. (Written. Passed by reference.)

Description

Allows the operator to move the cursor to any modifiable field in the current form (that is, to any field that is not Display Only and not Supervisor Only when the Supervisor Only flag is on) and to enter a value in that field only. The cursor is initially positioned at the current field and index. The current field name and index are updated in the workspace as the operator moves the cursor.

If the terminator is a function key not reserved for FMS, the form's function key UAR is called. The function key UAR may suppress the terminator, ignore it, or change it before subsequent processing occurs.

If the terminator is not a Previous Field terminator or a function key, the Form Driver checks the field for Response Required, Must Fill, and field completion UAR requirements. If any requirements are not fulfilled, the operator must continue input.

The Form Driver records in the workspace the value and terminator that the operator enters, and returns the input to your program. The operator can move about the form using the Next Field and Previous Field keys. The call ends when the operator modifies one field, presses the ENTER key, or types any function key not reserved for FMS. If the operator modifies a field, the Form Driver returns the status code FDV$_MOD instead of FDV$_SUC.

If the form contains scrolled areas, only the current scrolled line for each area is accessed by the call.

Status Codes

FDV$_ARG	Incorrect number of arguments.
FDV$_CAN	Call was terminated by a CANCL call.
FDV$_DSP	Form contains only display-only fields.
FDV$_INI	No workspace is defined.
FDV$_MOD	Field value in fldval has been modified by the operator. Otherwise, this code is the same as FDV$_SUC.
FDV$_NDS	Form is marked as being not displayed, so no input is possible.
FDV$_NFL	No form loaded in workspace.
FDV$_NOF	Form contains no fields.
FDV$_STR	Value being returned is too large for the variable allocated for it.
FDV$_SUC	Successful completion of the call.
FDV$_SYS	Form Driver encountered system error response.
FDV$_TCA	No terminal control area (TCA) is defined.
FDV$_TMO	Operator took longer to respond than allowed by the timeout value associated with the current terminal.

FDV$_UAR	UAR returned illegal code.
FDV$_UDP	UAR depth exceeded.
FDV$_UNF	UAR specified but not found.

5.21. Get All Field Values

FDV$GETAL ([fldval,fldtrm[,fldnam[,fldidx]]])

fldval	The values of all fields in the current form. The values are returned in the order specified in the form description. They consist of data characters but no field marker characters. If the operator does not enter a character for every position in the field, the Form Driver fills the empty positions with fill characters. (Written. Passed by descriptor.)
fldtrm	The field terminator that the operator entered to terminate input to the field. (Written. Passed by reference.)
fldnam	The name of the starting field. (Read. Passed by descriptor.)
fididx	The index of the starting field. (Read. Passed by reference.)

Description

Allows the operator to move the cursor to any nonscrolled fields in the current form that are modifiable (that is, to any nonscrolled fields that are not Display Only and not Supervisor Only when the Supervisor Only flag is on), and to enter values in those fields. This call normally positions the cursor at the first nonscrolled modifiable field but if you specify a field with the fldnam and fldidx arguments, input begins with that field instead.

The operator can move about the form using the Next Field and Previous Field keys. The call ends when the operator presses the Enter Form key or a non-FMS function key. The Form Driver processes each field terminator according to the description of the PFT call.

If the terminator is a function key not reserved for FMS, the form's function key UAR is called. The function key UAR may suppress the terminator, ignore it, or change it before subsequent processing occurs.

If the terminator is not a Previous Field terminator or a function key, the Form Driver checks the field for Response Required, Must Fill, and field completion UAR requirements. If any requirements are not fulfilled, the operator must continue input.

Call processing ends if an error occurs, or if the operator presses a function key that is not suppressed by a function key UAR.

If the operator presses the Enter Form key, but the form has nonscrolled fields with Response Required or Must Fill attribute requirements not fulfilled, or field completion UARs not satisfied, the Form Driver displays a message at the bottom of the screen, signals the operator, positions the cursor at the first field still requiring operator input, and awaits further input. (The operator is not restricted to entering data in these fields. The Form Driver moves the cursor only to direct the operator's attention to the fields.)

Upon completion of the form, the values of all nonscrolled fields (including display-only fields) are returned in fldval in the default field access order. The final field terminator and the modify flag status are returned as well.

If the form contains any scrolled areas, they are ignored by this call.

Status Codes

FDV$_ARG	Incorrect number of arguments.
FDV$_CAN	Call was terminated by a CANCL call.
FDV$_DSP	Form contains only display-only nonscrolled fields, or the field specified was display only.
FDV$_INI	No workspace is defined.
FDV$_MOD	At least one field has been modified by the operator.
FDV$_NDS	Form is marked as being not displayed, so no input is possible.
FDV$_NFL	No form loaded in workspace.
FDV$_NOF	Form contains no fields.
FDV$_STR	Value being returned is too large for the variable allocated for it.
FDV$_SUC	Successful completion of the call.
FDV$_SYS	Form Driver encountered system error response.
FDV$_TCA	No terminal control area (TCA) is defined.
FDV$_TMO	Operator took longer to respond than allowed by the timeout value associated with the current terminal.
FDV$_UAR	UAR returned illegal code.
FDV$_UDP	UAR depth exceeded.
FDV$_UNF	UAR specified but not found.

5.22. Get Data Line from Terminal

FDV$GETDL (value,fldtrm[,line[,prompt]])

value	The data line. (Written. Passed by descriptor.)
fldtrm	The field terminator that the operator entered to terminate input to the field. (Written. Passed by reference.)
line	The number of the line on which the operator's input is displayed. If you specify zero or omit this argument, the display occurs on the last line of the screen (24 or 14). (Read. Passed by reference.)
prompt	The data line text. Used as a prompt for the operator. (Read. Passed by descriptor.)

Description

Waits for the operator to type a line of text from the terminal.

The following points are important for you to note:

- This call does not require a workspace or TCA.
- The terminator returned is not saved as the current terminator.
- Any terminator is legal.

- The text returned has a length in the range of 0 to 132 characters.
- The text cannot be longer than the current width of the screen used for the input that is, 40, 66, 80, or 132 characters, depending on the screen width and the attributes of the line set by any form on the screen.
- If you specify text for the **prompt** argument, you reduce the size of the allowed input by the length of the prompt. The operator cannot delete the prompt.
- No function key or field completion UAR is called. Help is not available. The input editing functions available are the same as for input to a field.

If **line** has a value that is not zero, the value specifies the line on the screen to be used for the input. If **line** has a value of zero, the bottom line of the screen is used. The Form Driver clears the line prior to input and does not restore it after input is complete.

If the data line overwrites part of a form on the screen, the Form Driver may redisplay part or all of that form when your program issues the next PUT-type or GET-type call to it.

Status Codes

FDV$_ARG	Incorrect number of arguments.
FDV$_CAN	Call was terminated by a CANCL call.
FDV$_DLN	Argument **prompt** supplied more data than was required, and some data was discarded.
FDV$_STR	Value being returned is too large for variable allocated for it.
FDV$_SUC	Successful completion of the call.
FDV$_SYS	Form Driver encountered system error response.
FDV$_TMO	Operator took longer to respond than allowed by the timeout value associated with the current terminal.

5.23. Get Current Line of Scrolled Area

FDV$GETSC (fldnam,fldval[,fldtrm])

fldnam	A field name identifying the scrolled area. The field name need not specify an input field. (Read. Passed by descriptor.)
fldval	The field value. The value consists of data characters but no field-marker characters. If the operator does not enter a character for every position in the field, the Form Driver fills the empty positions with fill characters. (Written. Passed by descriptor.)
fldtrm	The field terminator that the operator entered to terminate input to the field. (Written. Passed by reference.)

Description

Positions the cursor at the first modifiable field of the current scrolled line within the scrolled area containing the named field. But if the previous call was a PFT call that processed a field terminator of FDV$K_FT_SPR (Scroll to Previous Line), the cursor is positioned at the last modifiable field in the line.

The Form Driver then allows the operator to enter data in the modifiable fields of the line, moving from one field to another either by pressing the Next Field and Previous Field keys, or by filling a field having the Autotab attribute.

If the terminator is a function key not reserved for FMS, the form's function key UAR is called. The function key UAR may suppress the terminator, ignore it, or change it before subsequent processing occurs.

If the terminator is not a Previous Field terminator or a function key, the Form Driver checks the field for Response Required, Must Fill, and field completion UAR requirements. If any requirements are not fulfilled, the operator must continue input.

The value of every field in the scrolled line is returned in **fldval.** When the processing of a terminator would cause the cursor to exit the line, the call is done. The call is also completed if the operator presses the Enter Form key or a function key.

The most recent field terminator code is returned.

Status Codes

FDV$_ARG	Incorrect number of arguments.
FDV$_CAN	Call was terminated by a CANCL call.
FDV$_DSP	Form contains only Display Only fields.
FDV$_FLD	Field does not exist.
FDV$_INI	No workspace is defined.
FDV$_MOD	Field value in **fldval** has been modified by the operator. Otherwise, this code is the same as FDV$_SUC.
FDV$_NDS	Form is marked as being not displayed, so no input is possible.
FDV$_NFL	No form loaded in workspace.
FDV$_NOF	Form contains no fields.
FDV$_NSC	Field named is not a field in a scrolled area.
FDV$_SUC	Successful completion of the call.
FDV$_SYS	Form Driver encountered system error response.
FDV$_TCA	No terminal control area (TCA) is defined.
FDV$_TMO	Operator took longer to respond than allowed by the timeout value associated with the current terminal.
FDV$_UAR	UAR returned illegal code.
FDV$_UDP	UAR depth exceeded.
FDV$_UNF	UAR specified but not found.

5.24. Return Illegal Terminators

FDVSILTRM (trmmod)

trmmod	A value determining whether illegal terminators are to be returned to your program, or treated as errors: 1 = Return illegal terminators.

	0 = Do not return illegal terminators. (Read. Passed by reference.)

Description

Allows your program to receive terminators that are normally illegal in certain contexts—for example, a Next Field terminator in the last field of a form or a Scroll Forward in a nonscrolled field. You can also restore the default state in which illegal terminators are not returned.

If your program issues ILTRM (1), any illegal terminator (listed below) from the current terminal is converted to a special terminator code and is treated as if it came from the pressing of a function key. The Form Driver sends the code to the form's function key UAR (if any), where the code can be rejected, converted to another terminator, or accepted. If there is no function key UAR, the illegal terminator ends input for the current call and is returned to your program.

Following is a list of illegal terminators, all marked by the characters ILG:

FDV$KJFT_ILG_NXT

K_FT_ILG_PRV

K_FT_ILG_ATB

K_FT_ILG_XBK

K_FT_JLG_XFW

K_FT_ILG_SFW

FDV$K_FT_ILG_SBK

Status Codes

FDV$_ARG	Incorrect number of arguments.
FDV$_CAN	Call was terminated by a CANCL call.
FDV$_SUC	Successful completion of the call.
FDV$_TCA	No terminal control area (TCA) is defined.
FDV$_VAL	The value of **trmmod.** is outside the allowed range.

5.25. Set Channel for Form Library File

FDV$LCHAN (channel)

channel	The logical I/O channel number for the form library. (Read. Passed by reference.)

Description

Specifies the current library logical channel. The current library channel is associated with the current terminal, so the terminal must be defined prior to execution of this call. Following the execution of LCHAN, the Form Driver uses the specified channel for any LOPEN or LCLOS call processing.

Your program normally issues an LCHAN before executing any other call that references the current library channel. The program can issue an LOPEN call without issuing an LCHAN first, however, if you choose to specify a channel in the LOPEN call. You can use LCHAN to switch from one open library to another open library.

The channels specified in the Form Driver calls ATERM, LCHAN, and LOPEN are strictly local to FMS and have no relationship to Logical Unit Numbers used by FORTRAN and BASIC. These channel numbers provide a means of reference only. The Form Driver keeps an association list of all logical channels currently in use by the application program. Logical terminal numbers and logical form library numbers must not conflict; that is, a logical terminal channel number cannot also be used as a logical form library channel number.

Status Codes

FDV$_ARG	Incorrect number of arguments.
FDV$_CAN	Call was terminated by a CANCL call.
FDV$__ICH	Logical channel specified was either in use or invalid.
FDV$_SUC	Successful completion of the call.
FDV$_TCA	No terminal control area (TCA) is defined.

5.26. Close Form Library

FDV$LCLOS

Description

Closes the form library associated with the current library channel. The current library channel is associated with the current terminal, so the terminal must be defined prior to the execution of this call.

Note that if a disk-resident form is displayed on the screen and you then issue the LCLOS call, Help forms cannot be accessed from that library.

Status Codes

FDV$_ARG	Incorrect number of arguments.
FDV$_CAN	Call was terminated by a CANCL call.
FDV$_FCH	Form library is already closed.
FDV$_ICH	Channel specified was invalid.
FDV$_SUC	Successful completion of the call.
FDV$_TCA	No terminal control area (TCA) isdefined.

5.27. Turn Terminal LED Off

FDV$LEDOF (ledno)

ledno	The number (in the range 1 to 4) of a LED to be turned off. (Read. Passed by reference.)

Description

Turns off the specified VT100 light-emitting diode (LED) of the current terminal if the current terminal is defined. If the current terminal is not defined, the call turns off the specified LED of the application program's default terminal instead. If the terminal is not a VT100-compatible terminal, the call is ignored but a success code is returned.

If LEDOF is called without a TCA, all the LEDs for the default terminal are turned off.

If at any time the operator presses the Refresh key, or if your program issues a RFRSH call, the LEDs are restored to their previous states.

Status Codes

FDV$_ARG	Incorrect number of arguments.
FDV$_CAN	Call was terminated by a CANCL call.
FDV$_SUC	Successful completion of the call.
FDV$_SYS	Form Driver encountered system error response.
FDV$_VAL	The value of **ledno** is outside the allowed range.

5.28. Turn Terminal LED On

FDV$LEDON (ledno)

Description

Turns on the specified VT100 light-emitting diode (LED) of the current terminal if the current terminal is defined. If the current terminal is not defined, the call turns on the specified LED of the application program's default terminal instead. Any other LEDs previously turned on for the default terminal are turned off. If the terminal is not a VT100 or a VT100-compatible terminal, the call is ignored but a success code is returned.

If at any time the operator presses the Refresh key, or if your program issues a RFRSH call, the LEDs are restored to their previous states.

Status Codes

FDV$_ARG	Incorrect number of arguments.
FDV$_CAN	Call was terminated by a CANCL call.
FDV$_SUC	Successful completion of the call.
FDV$_SYS	Form Driver encountered system error response.
FDV$_VAL	The value of **ledno** is outside the allowed range.

5.29. Load Form without Display

FDV$LOAD (frcnnam)

frmnam	The name of the form. (Read. Passed by descriptor.)

Description

Loads a binary form description into a workspace without displaying the form. Although the workspace is linked to a TCA, any form loaded by means of this call is marked as undisplayed and does not appear on the screen when this call is executed. Similarly, the form is not displayed if a RFRSH call is executed.

For a loaded but undisplayed form, all calls requiring operator action are illegal and return a status of FDV$_NDS. All other calls succeed, but where both the screen and workspace would normally be altered or updated by the call, only the workspace is altered — the screen is unaffected.

Status Codes

FDV$_ARG	Incorrect number of arguments.
FDV$_CAN	Call was terminated by a CANCL call.
FDV$_FCH	Form was not memory resident, and when the Form Driver attempted to search for it in a form library, the current library channel was not open.
FDV$_FNM	Binary form description could not be found either in the form library or in the list of memory-resident forms.
FDV$_FRM	Form description is invalid.
FDV$_IFU	Workspace cannot be loaded at this time because it is the workspace for a currently active UAR.
FDV$_INI	No workspace is defined.
FDV$_IOR	I/O error occurred while Form Driver was reading in the form from the form library. The I/O error code is recorded in the current state. You can obtain it by issuing the STAT call.
FDV$_IVM	Not enough virtual memory could be allotted for workspace.
FDV$_SUC	Successful completion of the call.
FDV$_TCA	No terminal control area (TCA) is defined.

5.30. Open Form Library

FDV$LOPEN (filspc[^channel])

filspc	The file specification for the form library you want to open. (Read. Passed by descriptor.)
channel	The logical I/O channel number for the form library. If this value is zero, the value you specified in the most recent LCHAN call for the current terminal remains in effect. (Read. Passed by reference.)

Description

Opens the form library associated with the channel you specify. If you omit the channel specification, the call assumes the current channel. (See also the description of the LCHAN call.)

You must issue this call before any other calls that fetch forms from the form library specified by filspc.

The channels specified in the Form Driver calls ATERM, LCHAN, and LOPEN are strictly local to FMS and have no relationship to Logical Unit Numbers used by FORTRAN and BASIC. These channel numbers provide a means of reference only. The Form Driver keeps an association list of all logical channels currently in use by the application program. Logical terminal numbers and logical form library

numbers must not conflict; that is, a logical terminal channel number cannot also be used as a logical form library channel number.

Status Codes

FDV$_ARG	Incorrect number of arguments.
FDV$_CAN	Call was terminated by a CANCL call.
FDV$_FLB	File specified was not a form library.
FDV$_FSP	File specification was invalid.
FDV$_ICH	Channel specified was either in use or invalid.
FDV$_JOL	Form Driver encountered an error while reading the form library (it reads the form library to verify that the file is a form library file).
FDV$_IOR	The Form Driver encountered an error while opening the form library.
FDV$_SUC	Successful completion of the call.
FDV$_TCA	No terminal control area (TCA) is defined.

5.31. Mark Form in Current Workspace as Not Displayed

FDVSNDISP

Description

Marks the form in the current workspace as being not displayed. The effect of this call is that on subsequent screen refreshes this form is not redisplayed, and subsequent GETs for the form are illegal. If the form is already on the screen, it remains there. To redisplay the form, your program must issue the

DISPW call. NDISP is useful if you are using more than one workspace.

NDISP can be particularly useful in some UARs. Normally, if a UAR needs to use another form to perform its task, it must attach a workspace, issue a DISP call to the workspace, and then detach the workspace when it is finished. Note that attaching a workspace is an expensive operation involving the allocation of memory from VMS. If the UAR does not detach the workspace, then the UAR working form is shown on every refresh operation thereafter.

Marking the form in the workspace as being not displayed is more efficient than the method described in the preceding paragraph. The workspace can be attached and loaded first, before any GETs. Then, when the UAR is activated, it can issue a DISPW to display the workspace when it needs it, and then perform an NDISP when it is finished. A Refresh operation at this time would bring back the original form, but not the UAR's working form.

No error occurs if the form is already marked as being not displayed or if there is no form in the workspace; such a form does not affect the screen.

Status Codes

FDV$_ARG	Incorrect number of arguments.
FDV$_CAN	Call was terminated by a CANCL call.

FDV$_INI	No workspace is defined.
FDV$_SUC	Successful completion of the call.
FDV$_TCA	No terminal control area (TCA) is defined.

5.32. Process Field Terminator

FDV$PFT ([fldtrm[,fldnam[,fldval[,nfldnam[,nfldidx]]]]])

fldtrm	The field terminator to be processed. (Read. Passed by reference.)
fldnam	A field name identifying a scrolled area. (Ignored in nonscrolled area.) (Read. Passed by descriptor.)
fldval	The field values to be displayed if the screen is scrolled during processing of a scrolling field terminator. (Ignored in nonscrolled area.) (Read. Passed by descriptor.)
nfldnam	The current field name after the call has been completed. (Written. Passed by descriptor.)
nfldidx	The current field index after the call has been completed. (Written. Passed by reference.)

Description

Processes a field terminator code. This call changes the current field or affects the current scrolled line in accordance with the terminator you supply with the call. If you omit **fldtrm** from the call, the Form Driver supplies the most recent terminator that the operator entered from the current terminal.

Note that the PFT call does not itself change the screen or move the cursor. It merely changes the current field for the workspace. Your program can then get the name of the new current field (from **nfldnam)** and issue a GET call specifying the new field name. The Form Driver then moves the cursor to the new field.

Terminators	**Action**
FDV$K_FT_NTR=0	Tests all nonscrolled modifiable fields to see if they satisfy the Response Required and Must Fill attributes and calls all field-validation routines for those fields. If all fields satisfy the criteria, the Form Driver returns the FDV$_SUC code to your program. If any fields do not satisfy the criteria, the first such field (in order of access) becomes the cur- rent field, and the Form Driver returns the FDV$_INC code.
FDV$K_FT_NXT=1	Makes the next modifiable field (in order of access) the current field. If the terminated field is the last modifiable field in a scrolled line, the terminator behaves like an FDV $K_FT_SNX. If the field is not in a scrolled area and there is no next modifiable field, the Form Driver returns the FDV $_JFN code.
FDV$K_FT_PRV=2	Makes the previous modifiable field (in order of access) the current field. If the field is the first field in a scrolled area, the terminator behaves like an FDV$K_FT_SPR. If the field is not in a scrolled area and there is no previous modifiable field, the Form Driver returns the FDV$_IFN code.

Terminators	Action
FDV$K_FT_ATB=3	(Autotab attribute) Behaves like an FDV$K_FT_NXT.
FDV$K_FT_XBK=4	Makes the first modifiable field preceding the scrolled area containing fldnam the current field. If there is no modifiable field preceding the scrolled area, the Form Driver returns the FDV$_JFN code.
FDV$K_FT_XFW=5	Makes the first modifiable field following the scrolled area containing fldnam the current field. If there is no modifiable field following the scrolled area, the Form Driver returns the FDV$_IFN code.
FDV$K_FT_SNX=6	Scrolls forward to next field (from Next Field or Autotab terminator in last field of scrolled area). Makes the next modifiable field in the scrolled area containing fldnam the current field. The area is scrolled up, and the new last line is filled with the values in fldval, if you supplied them in the call (as in a PUTSC call). If you omitted fldval, the Form Driver supplies default values.
FDV$K_FT_SPR=7	Scrolls backward to previous field (from Previous Field terminator in first field of scrolled area). Makes the previous modifiable field in the scrolled area containing fldnam the current field. The area is scrolled down, and the new first line is filled with the values in fldval, if you supplied them in the call (as in a PUTSC call). If you omitted fldval, the Form Driver supplies default values.
FDV$K_FT_SFW=8	Scrolls forward. Makes the first modifiable field in the scrolled area containing fldnam the current field. If the terminated scrolled line is the last in the scrolled area, the area is scrolled up, and the new last line is filled with the values in fldval, if you supplied them in the call (as in a PUTSC call). If you omitted fldval, the Form Driver supplies default values.
FDV$K_FT_SBK =9	Scrolls backward. Makes the first modifiable field in the scrolled area containing fldnam the current field. If the terminated scrolled line is the first in the scrolled area, the area is scrolled down, and the new first line is filled with the values in fldval, if you supplied them in the call (as in a PUTSC call).If you omitted fldval, the Form Driver supplies default values.

If the field terminator code is not listed above, the Form Driver returns the FDV$_UTK code to your program.

If the Form Driver returns the FDV$_IFN or FDV$_UTR status codes, the current field does not change, and **nfldnam** and **nfldidx,** if you specified them in the call, reflect this status.

Status Codes

FDV$_ARG	Incorrect number of arguments.
FDV$_CAN	Call was terminated by a CANCL call.
FDV$_DLN	Value argument supplied more data than was required, and some data was discarded.

FDV$_FLD	Field does not exist.
FDV$_IFN	Field terminator code cannot be processed in the context indicated.
FDV$_INI	No workspace is defined.
FDV$_NFL	No form loaded in workspace.
FDV$_NOF	Form contains no fields.
FDV$_NSC	Field named is not a field in a scrolled area.
FDV$_SUC	Successful completion of the call.
FDV$_SYS	Form Driver encountered system error response.
FDV$_TCA	No terminal control area (TCA) is defined.
FDV$_UTR	Field terminator code is invalid.

5.33. Output Value to Specified Field

FDV$PUT (fldval,fldnam[,fididxl)

fldval	The field value. The data passed must consist only of the characters to be displayed in the data positions of the field. Field-marker characters must not be passed. (Read. Passed by descriptor.) Note that the Form Driver does not check the validity of the data against the field picture.
fldnam	The field name. (Read. Passed by descriptor.)
fldidx	The field index. (Read. Passed by reference.)

Description

Records the value specified by **fldval** in the workspace, and, if the workspace is marked as displayed, updates the field on the screen with the new data. If **fldval** is shorter than the field, then **fldval** is justified as the field-justification attribute requires, and the remainder of the field is padded on either the right or left, according to that field attribute. The clear character pads the field on the screen, and the fill character pads the field in the workspace.

If **fldval** is null, the field is filled with its default value. If **fldval** is too long for the field specified, the **fldval** string is truncated on the right, and the field is filled in with the truncated value, from the leftmost portion of the output string. The status code FDV$_DLN is displayed if the Form Driver is in

Debug mode, but FDV$_SUC is returned to your program.

If a field having the date or time attribute is to be filled with a default value, and no default value is defined in the form description, the current date or time becomes the default.

If the field you specify is in a scrolled area, the field is displayed on the current scrolled line of that area.

Status Codes

FDV$_ARG	Incorrect number of arguments.
FDV$_CAN	Call was terminated by a CANCL call.

FDV$_DLN	Value argument supplied more data than was required, and some data was discarded.
FDV$_FLD	Field does not exist.
FDV$_INI	No workspace is defined.
FDV$_NFL	No form loaded in workspace.
FDV$_NOF	Form contains no fields.
FDV$_SUC	Successful completion of the call.
FDV$_SYS	Form Driver encountered system error response.
FDV$_TCA	No terminal control area (TCA) is defined.

5.34. Output Values to All Fields

FDV$PUTAL ([frmval])

frmval	The values for all fields. (Read. Passed by descriptor.)

Description

Takes data from **frmval,** records it in the workspace, and, if the workspace is marked as displayed, updates the screen with the new field values. You can alter all fields or all nonscrolled fields of the form with this call.

If the form contains any scrolled areas, they are ignored by this call provided **frmval** is specified. If you omit the **frmval** value, however, the scrolled areas are restored to their default values, and the current scrolled line of each scrolled area is reset to the first line of the area.

If **frmval** contains more data than is required to define every field, the excess data is discarded, and the Form Driver displays the status code of FDV$_DLN if Debug mode is in effect, but returns FDV$_SUC to your program. If **frmval** contains insufficient data to define every field, the remainder are defined by the default values for the fields.

If a field having the date or time attribute is to be filled with a default value, and no default value is defined in the form description, the current date or time becomes the default.

The order of the field values in **frmval** is specified in the form description.

Status Codes

FDV$_ARG	Incorrect number of arguments.
FDV$_CAN	Call was terminated by a CANCL call.
FDV$_DLN	Value argument supplied more data than was required, and some data was discarded.
FDV$_INI	No workspace is defined.
FDV$_NFL	No form loaded in workspace.
FDV$_NOF	Form contains no fields.
FDV$_SUC	Successful completion of the call.
FDV$_SYS	Form Driver encountered system error response.

FDV$_TCA	No terminal control area (TCA) is defined.

5.35. Output Default to Specified Field

FDV$PUTD (fldnam[,fldidx])

fldnam	The field name. (Read. Passed by descriptor.)
fldidx	The field index. (Read. Passed by reference.)

Description

Causes the default value, if any, to be restored to the specified field. If none is defined, the field is filled with fill characters in the workspace and with clear characters on the screen. The values are displayed only if the workspace is marked as displayed. PUTD duplicates a portion of the PUT function and is provided to support those languages that do not allow omission of arguments from a call.

If a field having the date or time attribute is to be filled with a default value, and no default value is defined in the form description, the current date or time becomes the default.

If the field you specify is in a scrolled area, the field default is restored for the current scrolled line only.

Status Codes

FDV$_ARG	Incorrect number of arguments.
FDV$_CAN	Call was terminated by a CANCL call.
FDV$_FLD	Field does not exist.
FDV$_INI	No workspace is defined.
FDV$_NFL	No form loaded in workspace.
FDV$_NOF	Form contains no fields.
FDV$_SUC	Successful completion of the call.
FDV$_SYS	Form Driver encountered system error response.
FDV$_TCA	No terminal control area (TCA) is defined.

5.36. Output Default Values to All Fields

FDV$PUTDA

Description

Causes the default values, if any, to be restored to all fields in a form. If none is defined for a field, the field is filled with clear characters on the screen and with fill characters in the workspace. Note that the default values are displayed only if the workspace is marked as displayed. This function duplicates a portion of the PUTAL function and is provided to support those languages that do not allow the omission of all arguments from a call.

If a field having the date or time attribute is to be filled with a default value, and no default value is defined in the form description, the current date or time becomes the default.

If the form contains any scrolled areas, the defaults are restored to the fields in the scrolled areas, and the current scrolled line of each area is reset to the first line of each area.

Status Codes

FDV$_ARG	Incorrect number of arguments.
FDV$_CAN	Call was terminated by a CANCL call.
FDV$_INI	No workspace is defined.
FDV$_NFL	No form loaded in workspace.
FDV$_NOF	Form contains no fields.
FDV$_SUC	Successful completion of the call.
FDV$_SYS	Form Driver encountered system error response.
FDV$_TCA	No terminal control area (TCA) is defined.

5.37. Output Line to Screen

FDV$PUTL (text[,line])

text	The line of text for the data line. (Read. Passed by descriptor.)
line	The number of the line on which the Form Driver displays the data line. If you specify zero, the display occurs on the last line of the screen (24 or 14). (Read. Passed by reference.)

Description

Displays text on the line specified by line; or if the line value is zero, on the last line of the screen. The line is always deleted before the text is displayed.

Normally, the last line is line 24 of the screen. If the screen is in 132-column mode, however, and the terminal lacks the Advanced Video Option, the last line is line 14.

If line specifies the last line of the screen, the Form Driver clears the line of text when the operator types the next character. If line is not zero, your program has to clear the line.

The text can be 40, 66, 80, or 132 characters, depending on the current screen size and the attributes of the line. If the message does not fit on the current screen, it is truncated and the status code of FDV $_DLN is reported if the Form Driver is in Debug mode, although FDV$_SUC is returned to your program. A message longer than 80 characters is truncated on a VT52 terminal.

On a VT100 with the advanced video option, the displayed line has the bold video attribute by default. If the terminal does not have the advanced video option, the line is displayed in the same video mode as the cursor (underline or reverse video), which the operator sets by using the VT100's Set-Up mode.

If you specified video attributes in an ADLVA call, those attributes are used instead.

If the terminal is a VT52, the line is displayed in normal video.

If line overwrites part of a form on the screen, the Form Driver may redisplay part or all of that form when your program issues the next PUT-type or GET-type call to it. This is true even if the line overwritten was blank or was specified only in the area to clear portion of the form.

Status Codes

FDV$_ARG	Incorrect number of arguments.

FDV$_CAN	Call was terminated by a CANCL call.
FDV$_DLN	Value argument supplied more data than was required, and some data was discarded.
FDV$_LIN	The **line** argument is invalid. It is either negative or greater than the number of lines that can be displayed on the screen (24 lines normally, or 14 if in 132-column mode and on a VT100 without the advanced video option).
FDV$_SUC	Successful completion of the call.
FDV$_SYS	Form Driver encountered system error response.

5.38. Output Data to Current Line of Scrolled Area

FDVSPUTSC (fldnam[,fldval])

fldnam	A field name identifying the scrolled area. (Read. Passed by descriptor.)
fldval	The field values. (Read. Passed by descriptor.)

Description

Outputs data to the scrolled area containing the field named in fldnam. The line in the scrolled area that is displayed is the current scrolled line for that area.

All fields on the current line are updated with fldval. If not enough data is supplied in fldval, the remaining fields are set to their default values, or cleared if there is no default. If too much data is supplied, the Form Driver truncates the data and reports the status code FDV$_DLN if Debug mode is in effect, although FDV$_SUC is returned to your program.

If a field having the date or time attribute is to be filled with a default value, and no default value is defined in the form description, the current date or time becomes the default.

The order of the fields in fldval is specified in the form description.

Status Codes

FDV$_ARG	Incorrect number of arguments.
FDV$_CAN	Call was terminated by a CANCL call.
FDV$_DLN	Value argument supplied more data than was required, and some data was discarded.
FDV$_FLD	Field does not exist.
FDV$_INI	No workspace is defined.
FDV$_NFL	No form loaded in workspace.
FDV$_NOF	Form contains no fields.
FDV$_NSC	Field named is not a field in a scrolled area.
FDV$_SUC	Successful completion of the call.
FDV$_SYS	Form Driver encountered system error response.

FDV$_TCA	No terminal control area (TCA) is defined.

5.39. Read Form into Memory

FDV$READ (frmnam,mloc,mlocsiz,frmsiz)

frmnam	The name of the form. (Read. Passed by descriptor.)
mloc	The area in which the form is to be stored. (Modified. Passed by descriptor.)
mlocsiz	The size of the memory buffer that begins with mloc. (Read. Passed by reference.)
frmsiz	The size of the form in bytes. (Written. Passed by reference.)

Description

Reads a form from a form library into the memory area that you specify and adds the form to the head of the list of memory-resident forms known to your program. Any subsequent references to the form get the form description from **mloc** rather than from the form library. The size of the form is returned in **frmsiz.** (See also the description of the DEL call.)

You can have forms with duplicate form names on the list of memory-resident forms, but only the form closest to the head of the list can be accessed.

Status Codes

FDV$_ARG	Incorrect number of arguments.
FDV$_CAN	Call was terminated by a CANCL call.
FDV$_FCH	Form was not resident, and when the Form Driver attempted to search for it in a form library, the current library channel was not open.
FDV$_FNM	Binary form description could not be found in the form library.
FDV$_FRM	Form description is invalid.
FDV$_IBF	Area not large enough to hold the form.
FDV$_IOR	I/O error occurred while Form Driver was reading in the form from the form library. The I/O error code is recorded in the current state. You can obtain it by issuing the STAT call.
FDV$_SUC	Successful completion of the call.
FDV$_TCA	No terminal control area (TCA) is defined.

5.40. Return Value for Specified Field

FDVSRET (fldval,fldnam[,fldidx])

fidval	The field value consisting of data characters but no field-marker characters. If the operator does not enter a character for every position in the field, the Form Driver fills the empty positions with fill characters. (Written. Passed by descriptor.)
fldnam	The field name. (Read. Passed by descriptor.)

fldidx	The field index. (Read. Passed by reference.)

Description

Returns the value of the field you specify from the current workspace. The data returned by the RET call is data already accepted from the operator by a previous GET-type call, data displayed by a previous PUT-type call, or data present by default. Note that unlike the GET call, RET accepts no input from the operator.

Display-only fields are returned by this call.

Status Codes

FDV$_ARG	Incorrect number of arguments.
FDV$_CAN	Call was terminated by a CANCL call.
FDV$_FLD	Field does not exist.
FDV$_INI	No workspace is defined.
FDV$_NFL	No form loaded in workspace.
FDV$_NOF	Form contains no fields.
FDV$_STR	Value being returned is too large for the variable allocated for it.
FDV$_SUC	Successful completion of the call.
FDV$_TCA	No terminal control area (TCA) is defined.

5.41. Return Values for All Fields

FDV$RETAL (frmval)

frmval	The values for all fields. The values returned consist only of the data character positions in the fields. No field-marker characters are returned. If data characters do not fill a field, the Form Driver fills the remainder of the field with the fill character. (Written. Passed by descriptor.)

Description

Returns the values of all nonscrolled fields from the current workspace. The order in which the fields are returned is specified in the form description. The data returned by the RETAL call is data already accepted from the operator by a previous GET-type call, data displayed by a previous PUT-type call, or data present by default. Note that unlike the GETAL call, RETAL accepts no input from the operator.

Display-only fields are among those returned by this call.

Status Codes

FDV$_ARG	Incorrect number of arguments.
FDV$_CAN	Call was terminated by a CANCL call.
FDV$_INI	No workspace is defined.
FDV$_NFL	No form loaded in workspace.
FDV$_NOF	Form contains no fields.

FDV$_STR	Value being returned is too large for the variable allocated for it.
FDV$_SUC	Successful completion of the call.
FDV$_TCA	No terminal control area (TCA) is defined.

5.42. Return Current Context

FDV$RETCX (atca,awksp,frmnarn,uarval,curpos,fldtrm,insovrthlpnimi)

atca	The address of the current terminal control area. If this location is zero, no TCA is defined. (Written. Passed by reference.) Not all high-level languages are capable of handling addresses.
awksp	The address of the current workspace. If this location contains a zero, no workspace is defined. (Written. Passed by reference.) Not all high-level languages are capable of handling addresses.
frmnam	The name of the form being processed. (Written. Passed by descriptor.)
uarval	The value of the associated UAR text, if one is defined. (Written. Passed by descriptor.)
curpos	The cursor position within the current field, if any. The cursor position is 1 for the leftmost data character in the field, 2 for the next data character to the right, n for the rightmost character in the field, and *n* + 1 for the character position to the immediate right of the rightmost data character (the hanging cursor position). Field-marker characters are not counted by the cursor. The range of the cursor,1 to *n* + 1, is limited to the number of data characters in the field plus 1. (Written. Passed by reference.) For fixed-decimal fields, the range of the cursor is 1 to *n* + 2, because the decimal point is counted even though it is not a data character. This allows the cursor to be positioned on the decimal point, in the hanging cursor position for the left hand part of the field. The **curpos** argument is always nonzero when a UAH is called during field processing (a field completion UAR, function key UAR, or help UAR).This argument can be zero if the RETCX call is executed when not in a UAR doing processing for a field. Zero means that the default position will be used for the next field access. The **curpos** argument is not always zero outside **UAR** processing for a field. If your program has previously issued an **AFCX** call on the current field, setting a nonzero **curpos,** then that nonzero value will be reported.
fldtrm	The field terminator that the operator last entered either to terminate input to a field or to respond to the execution of a WAIT call. (Written. Passed by reference.)
insovr	A value indicating whether Insert or Overstrike mode is in effect for a field. (Written. Passed by reference.)

	1 = Default 2 = Insert mode 2 = Overstrike mode The **insovr** argument is always nonzero when a UAR is called during field processing (a field completion UAR,function key UAR, or help UAR). This argument can be zero if the RETCX call is executed when a UAR is processing for a field. Zero means that the default position will be used for the next field access. The **insovr** argument is not always zero outside UAR processing for afield. If your program has previously issued an AFCX call on the current field, setting a nonzero **insovr**, then that nonzero value will be reported.
hipnum	A value equal to the number of times the operator has pressed the Help key for the current field. (Written. Passed by reference.)

Description

Returns the current context of the Form Driver as defined above. Your program can issue this call from a user action routine to determine the context in which the UAR is called, although use of RETCX is not limited to UARs.

Status Codes

FDV$_ARG	Incorrect number of arguments.
FDV$_CAN	Call was terminated by a CANCL call.
FDV$_J3TR	Value being returned is too large for the variable allocated for it.
FDV$_SUC	Successful completion of the call.

5.43. Return Named Data by Index

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

nmdidx	The Named Data index. (Read. Passed by reference.)
nmdval	The Named Data text. (Written. Passed by descriptor.)
nmdnam	The name of the Named Data. (Written. Passed by descriptor.)

Description

Returns the Named Data text you specify by index (rather than by name). This call also returns the Named Data name.

Status Codes

FDV$_ARG	Incorrect number of arguments.
FDV$_CAN	Call was terminated by a CANCL call.
FDV$_DNM	No Named Data is associated with the specified index.

FDV$_INI	No workspace is defined.
FDV$_NFL	No form loaded in workspace.
FDV$_STR	Value being returned is too large for the variable allocated for it.
FDV$_SUC	Successful completion of the call.
FDV$_TCA	No terminal control area (TCA) is defined.

5.44. Return Named Data by Name

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

nmdnam	The Named Data name. (Read. Passed by descriptor.)
nmdval	The Named Data text. (Written. Passed by descriptor.)
nmdidx	The Named Data index. (Written. Passed by reference.)

Description

Returns the Named Data text you specify by name (rather than by index). This call also returns the Named Data index.

Status Codes

FDV$_ARG	Incorrect number of arguments.
FDV$_CAN	Call was terminated by a CANCL call.
FDV$_DNM	No Named Data is associated with the specified index.
FDV$_INI	No workspace is defined.
FDV$_NFL	No form loaded in workspace.
FDV$_SUC	Successful completion of the call.
FDV$_TCA	No terminal control area (TCA) is defined.

5.45. Return Form Line

FDV$RETFL (fline,value,linlen[,type])

line	The number of the form line to be returned. (Read. Passed by reference.)
value	The image of the line you request. (Written. Passed by descriptor.)
linlen	The length of the **value** line in bytes. (Written. Passed by reference.)
type	The type of output you want:1 for current terminal image (including, for example, escape sequences for video) and 0 for line printer image. If type has a value of 0, the Form Driver makes the following correspondences. (Read. Passed by reference.) 1. All video attributes are ignored. 2. If the line indicated is double width, it is returned with each text item or field item centered in the area it would have occupied.

	3. If the line indicated is double size, then the first line is returned as a double-width line, and the second line is returned as a line of blanks. 4. If the line contains any line-drawing graphics, they are converted to standard ASCII characters: • The horizontal bar graphics are converted to ASCII dash characters (-). • Vertical bar graphics are converted to ASCII vertical bar characters (I). • All intersection graphics and comer graphics are converted to ASCII plus characters (+). All other characters in alternate character sets remain untranslated.

Description

Returns the form line you specify with the line argument. Usually, this is one of the lines you would see if your program issued a RFRSH call, although your program can issue RETFL to display lines from loaded, but undisplayed, forms as well.

If the current terminal has any attached workspaces with undisplayed forms, they are normally ignored by this call. But if undisplayed forms are the only forms in the attached workspaces, they are all included in generating the line image. Thus, your program can use undisplayed forms for report formatting purposes.

When using multiple workspaces, a call to RETFL returns the image of a line as it would appear on the screen after a RFRSH. More than one form may contribute to the line if forms overlap, and the last form displayed does not clear the line.

If type has a value of 1, the line image returned includes escape sequences and control characters to present an exact image of the screen if it were to be displayed on the same kind of terminal as the current terminal. The image so returned can be stored in a file and displayed later, or output to an intelligent printer that understands the same control sequences as the terminal.

Since the length of such an image can easily extend beyond 132 characters when there are many fields and text blocks on the line (especially if they specify varying video attributes and character sets), the buffer used has a capacity of 4000 bytes, which should be sufficient for all but multiple overlaid forms on a single line. If the buffer overflows, the error FDV$_LLI is returned. Saving, and later displaying, a very long line may cause problems due to RMS or VMS restrictions on file record size or I/O record sizes.

Status Codes

FDV$_ARG	Incorrect number of arguments.
FDV$_CAN	Call was terminated by a CANCL call.
FDV$_INI	No workspace is defined.
FDV$_LIN	Call specifies that some line not on the screen was requested.
FDV$_LLI	The Form Driver’s internal buffer was not large enough to store the line image requested. The line image returned is truncated.
FDV$_STR	Value being returned is too large for the variable allocated for it.

FDV$_SUC	Successful completion of the call.
FDV$_TCA	No terminal control area (TCA) is defined.

5.46. Return Current Field Name

FDV$RETFN (fldnaml.fldidx])

fldnam	The field name. (Written. Passed by descriptor.)
fididx	The field index. (Written. Passed by reference.)

Description

Returns the current field name and index from the current workspace. If the field is not indexed, RETFN returns an index value of zero. If there is no current field, the Form Driver returns a null string of characters for the field name.

Status Codes

FDV$_ARG	Incorrect number of arguments.
FDV$_CAN	Call was terminated by a CANCL call.
FDV$_INI	No workspace is defined.
FDV$_NFL	No form loaded in workspace.
FDV$_NOF	Form contains no fields.
FDV$_STR	Value being returned is too large for the variable allocated for it.
FDV$_SUC	Successful completion of the call.
FDV$_TCA	No terminal control area (TCA) is defined.

5.47. Return Field Names in Order

FDV$RETFO (fldnum,fldnam,fldidx)

fldnum	The nth field in the form, where *n* includes the number of any identically named indexed fields present. (Read. Passed by reference.)
fldnam	The name of the field corresponding to **fldnum.** (Written. Passed by descriptor.)
fldidx	The field index corresponding to **fldnum.** (Written. Passed by reference.)

Description

Returns the name and index of the nth field in the form where n includes the number of any identically named indexed fields present. If you want the fifth field in the form (n = 5), it could have a unique name, or be, for example, FIELD1 indexed down to the fifth field called FIELD1.

The field names can be in scrolled areas, but a field name in a scrolled area is returned only once, unless the field also happens to be indexed.

Status Codes

FDV$_ARG	Incorrect number of arguments.
FDV$_CAN	Call was terminated by a CANCL call.
FDV$_ELD	Field does not exist, or index value is invalid for field.
FDV$_INI	No workspace is defined.
FDV$_NFL	No form loaded in workspace.
FDV$_NOF	Form contains no fields.
FDV$_SUC	Successful completion of the call.
FDV$_TCA	No terminal control area (TCA) is defined.

5.48. Return Length of Specified Field

FDV$RETLE (fldlen,fldnam[,fldidx])

fldlen	The length of the field. The length is defined as the number of data positions in the field. The number of field-marker characters on the field has no effect in the determination of the length of the field. (Written. Passed by reference.)
fldnam	The field name. (Read. Passed by descriptor.)
fldidx	The field index. (Read. Passed by reference.)

Description

Returns the length of the field you specify. The length of a field is the number of data characters in the field exclusive of any field-marker characters.

Status Codes

FDV$_ARG	Incorrect number of arguments.
FDV$_CAN	Call was terminated by a CANCL call.
FDV$_DLN	Value argument supplied more data than was required, and some data was discarded.
FDV$_FLD	Field does not exist.
FDV$_ENI	No workspace is defined.
FDV$_NFL	No form loaded in workspace.
FDV$_NOF	Form contains no fields.
FDV$_SUC	Successful completion of the call.
FDV$_TCA	No terminal control area (TCA) is defined.

5.49. Refresh Screen

FDV$RFRSH

Description

Redisplays all forms currently marked as being displayed on the screen. This operation is identical to the one initiated by the operator's pressing of the Refresh key. If several forms are on the screen, they are redisplayed in the order that their workspaces were attached, except that the current work-space's form is always displayed last.

A screen refresh also restores the keypad mode. In addition, the refresh operation restores the terminal LEDs to the state they were in before the refresh occurred.

Status Codes

FDV$_ARG	Incorrect number of arguments.
FDV$_CAN	Call was terminated by a CANCL call.
FDV$_FCH	Form was not resident, and when the Form Driver attempted to search for it in a form library, the current library channel was not open.
FDV$_INI	No workspace is defined.
FDV$_SUC	Successful completion of the call.
FDV$_SYS	Form Driver encountered system error response.
FDV$_TCA	No terminal control area (TCA) is defined.

5.50. Set Screen Width

FDV$SCR_WIDTH (width)

width	An integer specifying the current width of the screen; must be either 80 or 132. (Read. Passed by reference.)

Description

Informs the Form Driver that your program has changed the width of the screen from the value last known to the Form Driver. Your program must also inform the operating system when it changes the screen width. The Form Driver does not change the screen or inform the operating system of screen width changes as a result of this call. However, the Form Driver always informs the operating system when other calls to the Form Driver change the screen width. Your program can query the operating system at any time for this information.

Status Codes

FDV$_ARG	Incorrect number of arguments.
FDV$_CAN	Call was terminated by a CANCL call.
FDV$_SUC	Successful completion of the call.
FDV$_TCA	No terminal control area (TCA) is defined.
FDV$_VAL	Width was not 80 or 132.

5.51. Signal Operator

FDV$SIGOP

Description

Signals the operator from the application program. Depending on the current signal mode for the terminal, either the terminal bell is rung or the video of the terminal is reversed until the operator next types a valid character (any character that does not generate another Form Driver signal). See also the description of the SSIGQ call.

This signaling is automatically performed prior to each error message issued by the Form Driver.

Status Codes

FDV$_ARG	Incorrect number of arguments.
FDV$_CAN	Call was terminated by a CANCL call.
FDV$_SUC	Successful completion of the call.
FDV$_SYS	Form Driver encountered system error response.
FDV$_TCA	No terminal control area (TCA) is defined.

5.52. Set Keypad to Application Mode

FDV$SPADA (mode)

mode	A value determining the keypad mode: • If **mode** contains 0, keypad = numeric mode. • If **mode** contains 1, keypad = application mode. Any other values are erroneous. (Read. Passed by reference.)

Description

Sets the terminal keypad mode. In numeric mode, the terminal keypad keys act as normal keys, returning the characters inscribed on them. When the keypad is in application mode, the keypad keys act as field terminator keys. The Form Driver resets the keypad of the current terminal to the selected mode whenever a Refresh operation occurs.

If no current terminal is in effect (TCA not defined), the default terminal is used in this call. Prior to the application making this call, the status of the keypad is determined by its VMS status. See the SET and SHOW TERMINAL commands in the *VMS Common Run-Time Library Manual*

Status Codes

FDV$_ARG	Incorrect number of arguments.
FDV$_CAN	Call was terminated by a CANCL call.
FDV$_SUC	Successful completion of the call.
FDV$_SYS	Form Driver encountered system error response.
FDV$_VAL	Width was not 80 or 132.

5.53. Turn Supervisor-Only Mode Off

FDVSSPOFF

Description

Sets the supervisor-only mode flag to Off. Following this call, the operator can alter fields marked as Supervisor Only in the form descriptions. The supervisor-only flag is altered only for the current terminal. There is a separate Supervisor Only flag for each terminal.

The supervisor-only flag is set to On when the terminal is first attached.

Status Codes

FDV$_ARG	Incorrect number of arguments.
FDV$_CAN	Call was terminated by a CANCL call.
FDV$_SUC	Successful completion of the call.
FDV$_TCA	No terminal control area (TCA) is defined.

5.54. Turn Supervisor-Only Mode On

FDV$SPON

Description

Sets the supervisor-only mode flag to be set to On. Following this call, fields marked as Supervisor Only in the form descriptions are treated as display only fields. The Supervisor Only flag is altered only for the current terminal. There is a separate Supervisor Only flag for each terminal.

The supervisor-only flag is set to On when the terminal is first attached.

Status Codes

FDV$_ARG	Incorrect number of arguments.
FDV$_CAN	Call was terminated by a CANCL call.
FDV$_SUC	Successful completion of the call.
FDV$_TCA	No terminal control area (TCA) is defined.

5.55. Set Signal to Quiet Mode

FDV$SSIGQ (sigmd)

sigmd	The signal mode value. (Read. Passed by reference.) 0 = Bell 1 = Reverse video

Description

Specifies the signal mode for the current terminal. If the signal mode is 0, the terminal bell is rung when you later issue the SIGOP call or the Form Driver issues any error message. If the mode is 1, the screen video is reversed when the signal occurs and automatically reverts back to the original video mode when the operator types the next valid character. See also the description of SIGOP.

If the signal mode is 1, and the Form Driver does not know what the screen video attribute of the terminal is, the Form Driver sets the terminal to normal video (white characters on black background).

The Form Driver knows the screen video attribute from then on regardless of any changes caused by subsequent form displays.

If the terminal is a VT52, the terminal bell is the signaling mode regardless of the mode setting. Attempts to specify video reversal as the signal mode for VT52-compatible terminals are ignored.

Status Codes

FDV$_ARG	Incorrect number of arguments.
FDV$_CAN	Call was terminated by a CANCL call.
FDV$_SUC	Successful completion of the call.
FDV$_TCA	No terminal control area (TCA) is defined.

5.56. Specify Status Reporting Variables

FDV$SSRV ([status,iostat]])

status	The value **of** the general status. This address becomes the general status reporting variable. (Written. Passed by reference.)
iostat	The value of the I/O status. This address becomes the I/O status reporting variable. (Written. Passed by reference.)

Description

Records the addresses of two variables in the current terminal's TCA:

- The address of a variable in which each subsequent call's I/O status is to be recorded
- The address of a variable in which each subsequent call's normal status is to be recorded

Following the execution of any call, if either address is not location 0, the appropriate call status is stored in the status variable. You can use this call to set up automatic status reporting instead of using the STAT call or VMS status returns.

The status variables must be 32-bit integers on all VAX systems.

It is the application program's responsibility to ensure that after it issues this call, the addresses specified remain valid until the call is issued again specifying zeros for addresses. When you specify zeros as addresses (or when you do not specify any arguments), further status reporting is discontinued.

Status Codes

FDV$_ARG	Incorrect number of arguments.
FDV$_CAN	Call was terminated by a CANCL call.
FDV$_SUC	Successful completion of the call.
FDV$_TCA	No terminal control area (TCA) is defined.

5.57. Return Status from Last Call

FDV$STAT (status,iostat])

status	The value of the general status. (Written. Passed by reference.)
iostat	The value of the I/O status. (Written. Passed by reference.)

Description

Returns the status code for the previous call. Note that a STAT call following a previous STAT call returns the result of the previous STAT. That result is almost always success.

Status Codes

FDV$_ARG	Incorrect number of arguments.
FDV$_CAN	Call was terminated by a CANCL call.
FDV$_SUC	Successful completion of the call.

5.58. Set Current Terminal

FDV$STERM (tea)

tea	The name of a terminal control area. (Modified. Passed by descriptor.)

Description

Makes a specified attached terminal (as indicated by its terminal control area) the current terminal. Your program must have previously attached the terminal with an ATERM call with the TCA specified.

Changing the current terminal also causes the current workspace to be changed to the workspace most recently associated with the new current terminal. If no workspace is attached to that terminal, then after the execution of this call the current workspace is undefined.

Status Codes

FDV$_ARG	Incorrect number of arguments.
FDV$_CAN	Call was terminated by a CANCL call.
FDV$_SUC	Successful completion of the call.
FDV$_TCA	No terminal control area (TCA) is defined.

5.59. Set Field Entry Timeout

FDV$STIME (time)

time	The number of seconds the Form Driver waits for the operator to respond to a GET-type call. This parameter is optional, and defaults to 0.(Read Passed by reference.)

Description

Specifies the number of seconds the Form Driver waits for the operator to respond to a GET-type call. Execution of this call cancels the effect of any previous STIME call for the current terminal. A negative

or zero time value causes the Form Driver to wait indefinitely for input (the default). A separate STIME is associated with each terminal.

After an STIME call, the Form Driver resets the timeout value for each character in a field. Thus, if afield has ten characters in it and the timeout value is 15 seconds, the operator has 15 seconds to respond with the first character and 15 seconds to respond with each of the other nine characters in the field.

Status Codes

FDV$_ARG	Incorrect number of arguments.
FDV$_CAN	Call was terminated by a CANCL call.
FDV$_SUC	Successful completion of the call.
FDV$_TCA	No terminal control area (TCA) is defined.

5.60. Set Current Workspace

FDVSSWKSP (wksp)

wksp	The form workspace location. (Modified. Passed by descriptor.)

Description

Makes the attached workspace you specify the new current workspace. If the workspace you specify is associated with a different terminal, the current terminal is changed as well. Your program must have previously attached the specified workspace to a terminal TCA by issuing an AWKSP call.

Status Codes

FDV$_ARG	Incorrect number of arguments.
FDV$_CAN	Call was terminated by a CANCL call.
FDV$_INI	No workspace is defined.
FDV$_SUC	Successful completion of the call.
FDV$_SYS	Form Driver encountered system error response.

5.61. Set Terminal Channel

FDVSTCHAN (channel)

channel	The number of a physical **I/O** channel (not a logical I/O channel) to be associated with the current terminal.(Read. Passed by reference.)

Description

Specifies a physical terminal channel to be used for the current terminal. When your program issued ATERM, the Form Driver allocated a physical channel to correspond to the logical channel specified. TCHAN specifies a physical channel different from the one allocated by ATERM.

TCHAN requires that the TCA be attached before your program issues this call.

The previous physical channel is released when your program issues this call. The logical channel number of the previous channel (the channel number specified in the ATERM call) is also released.

Note

If your program issues TCHAN and later detaches the associated TCA, the terminal is not released. Any channel specified by means of the TCHAN call must be released by your program.

Status Codes

FDV$_ARG	Incorrect number of arguments.
FDV$_CAN	Call was terminated by a CANCL call.
FDV$_ICH	Logical channel specified was either in use or invalid.
FDV$_SUC	Successful completion of the call.
FDV$_TCA	No terminal control area (TCA) is defined.

5.62. Set up User Refresh Routine

FDV$USER_REFRESH ([rfr_address])

rfr_address	Address of a user-supplied routine to refresh part of the terminal screen. If this argument is specified, all subsequent Form Driver refresh operations will call the user-supplied routine first. If this argument is null or not specified, no user refresh routine will be called on subsequent Form Driver refresh operations.

Description

Helps a program maintain part of the terminal screen independent of the Form Driver, when the Form Driver normally overwrites part or all of the screen. For example, when the Form Driver must perform a refresh operation for the current terminal, the terminal's screen is first cleared and set to the proper width and background. Then all the workspaces marked as displayed are redisplayed. If your program is maintaining part of the screen, the refresh operation's screen clear automatically deletes your program's part from the screen.

When the Form Driver refreshes the screen, it calls your refresh routine, if one has been supplied in a call to USER REFRESH. Your routine should clear and write its own part of the screen, call CLEAR_VA, if necessary, and then return. The Form Driver then redisplays the displayed workspaces. This allows the refresh function to affect both your program's screen area and the Form Driver's area.

The Form Driver calls your refresh routine in four circumstances:

- Your program calls RFRSH.
- The operator presses the Refresh key during data entry.
- Help processing or UAR processing caused some part of the Form Driver's screen to be overlaid. That is, a form marked as displayed was overlaid by a help form or by a form displayed by a Field Completion UAR, Function Key UAR, or Pre-Help or Post-Help UAR.

  Prior to returning to normal data entry after the help or UAR sequence, the Form Driver calls your refresh routine and then redisplays the required forms. If a help form or UAR form does not overlay a displayed form, the Form Driver does not call your refresh routine. You should design your program so that if a help form or a UAR action overlays your program's screen area, it should also overlay the Form Driver's screen area.

- The terminal width is changed when a new form is displayed by using a CDISP, DISP, or DISPW call, or by using a help form display operation.

The Form Driver calls your refresh routine as if it were a UAR. The refresh routine behaves exactly like a UAR, except that it must not change the terminal width. The Form Driver restores the current terminal, workspace, and field.

An application program should make a call to USER-REFRESH specifying a user routine before starting the separate screen display. A second call should be made to USER-REFRESH without any argument when your program has completed its separate screen display.

Status Codes

FDV$_ARG	Incorrect number of arguments.
FDV$_CAN	Call was terminated by a CANCL call.
FDV$_SUC	Successful completion of the call.
FDV$_TCA	No terminal control area (TCA) is defined.

5.63. Wait for Operator

FDV$WAIT ([fldtrm])

fldtrm	The returned field terminator that the operator entered to terminate the wait condition. (Written. Passed by reference.)

Description

Waits until the operator signals to proceed by pressing any terminator key. This call allows the Form Driver to synchronize the application program with the pace of the operator.

If the terminator is a function key not reserved for FMS, and if a form is loaded in the current workspace and has a function key UAR, the Form Driver calls that UAR. The function key UAR may suppress the terminator, ignore it, or change it before subsequent processing occurs. When a terminator is accepted, it is recorded in the workspace as the most recent terminator entered.

Status Codes

FDV$_ARG	Incorrect number of arguments.
FDV$_CAN	Call was terminated by a CANCL call.
FDV$_INI	No workspace is defined.
FDV$_SUC	Successful completion of the call.
FDV$_SYS	Form Driver encountered system error response.
FDV$_TCA	No terminal control area (TCA) is defined.
FDV$_TMO	Operator took longer to respond than allowed by the timeout value associated with the current terminal.
FDV$_UAR	UAR returned illegal code.
FDV$_UDP	UAR depth exceeded.
FDV$_UNF	UAR specified but not found.

Appendix A. VAX FMS Form Driver Calls

A.1. VAX Language-Independent Notation

Form Driver routines are invoked according to rules specified in the VAX Procedure Calling and Condition Handling Standard.

Form Driver routines can be invoked as subroutines or as functions:

As a subroutine	**CALL FDV$xxx (parameter1, parameterˆ,...)**
As a function	**VMS_stat.wlc.v = FDV$xxx (parameter1, parameter2,...)**

Access type, data type, passing mechanism, and parameter form are assigned to each parameter in a prescribed order:

<parameter-name>.<access typexdata 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.r41dnam,rt.dxl[,fldidx.rLr]

The notation for each parameter is explained below. Note that every Form Driver call returns a VMS status code in the form **VMS_stat.wlc.v**.

Parameter	**<access type>**	**<data type>**	**<passing mechanism>**	**<parameter form>**
fldval	w Write-only access	t Character-coded text string	d By descriptor	xl Fixed-length or dynamic string descriptor
fldtrm	w Write-only access	1 Longword integer (signed)	r By reference	-
fldnam	r Read-only access	t Character-coded text string	d By descriptor	xl Fixed-length or dynamic string descriptor
fldidx	r Read-only access	1 Longword integer (signed)	r By reference	-

A.2. Procedure Parameter Notation For Form Driver Calls

FMS uses a subset of the VAX procedure parameter notation. The following table explains the notation used for access type, data *type*, passing mechanism, and parameter form.

Notation	**<access type>**	**Comments**
r	Read-only access	Parameters for input
w	Write-only access	Parameters for output

Notation	<access type>	Comments
m	Modify access	Parameters for both input and output

Notation	<data type>
a	Virtual address
1	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

Notation	<parameter form>
a	Array reference or descriptor
xl	Fixed-length or dynamic string descriptor

Table A.1. VAX FMS Form Driver Calls

Call	Procedure Parameter Notation	
ADLVA	**FDV$ADLVA** (video.ml.r)	
	video	video attributes code of data line
	Alters the data line video attributes. You can specify Blink, Bold, Reverse, and/or Underline.	
AFCX	**FDV$AFCX** (insovr.rl.r,curpos.rl.r[,fldnam.rt.dx1Lfldidx.rLr]])	
	insovr	Insert/Overstrike mode of field
	curpos	cursor position within field
	fldnam	field name
	fldidx	field index
	Alters the default field context of the specified field. You can specify Insert or Overstrike mode and cursor position in the field before any GET operation involving that field.	
AFVA	**FDV$AFVA** (video.ml.r[tfidnam«rtdxl[,fldidx.rl.r]])	
	video	video attributes code for field
	fldnam	field name
	fldidx	field index
	Alters the field video attributes.	
ATERM	**FDV$ATERM** (tca.mLda,size.rLr,channeLrLr[,trmnal»rt.dxl [,faketrmtyp.rt.dxl[,optionsj*l.r]]])	

Call	Procedure Parameter Notation	
	or **FDV$ATERM** (tca.mt.dx1,size.rl.r,channel.rl.r[,trmnal.rLdxl [,faketrmtyp.rt.dxl[,options.rl.r]]])	
	tea	terminal control area
	size	size
	channel	logical I/O channel number for terminal
	trmnal	name of terminal
	faketrmtyp	name of terminal used for batch processing
	options	integer specifying Form Driver options
	Attaches a terminal to the Form Driver for processing forms over a specific, logical I/O channel, names a TCA for that terminal, and specifies the size of the TCA.	
AWKSP	**FDV$AWKSP** (wksp.ml.da,size.rl.r) or **FDV$AWKSP** (wksp.rtdxl,size.rLr)	
	wksp	form workspace
	size	estimate of workspace size
	Attaches a form workspace to a list of workspaces associated with the current TCA, specifies the size in bytes, and establishes that workspace as the current workspace.	
BELL	**FDV$BELL** Rings the terminal bell.	
CANCL	**FDV$CANCL** Cancels any other call presently being processed on the current terminal.	
CDISP	**FDV$CDISP** (frmnam.rt.dxl[,offset.rLr])	
	frmnam	form name
	offset	number controlling placement of form on screen
	Clears the screen and displays a form. The display position may be offset from the original form description.	
CLEAR	**FDV$CLEAR** ([line[,linecnt]])	
	line	line number of first line to clear
	linecnt	number of lines to clear
	Clears the entire screen unless otherwise specified with the arguments.	
CLEARJVA	**FDV$CLEAR_VA** Clears the screen video attributes.	
DEL	**FDV$DEL** (frmnam.rt.dx1)	
	frmnam	form name

Call	Procedure Parameter Notation	
	Deletes a form from the list of memory-resident forms.	
DFKBD	**FDV$DFKBD** (defkbd.rw.da,kbdnum.rl.r)	
	defkbd	array of key functions and key codes
	kbdnum	number of pairs of key functions and associated key codes in defkbd array
	Redefines the FMS keypad function keys.	
DISP	**FDV$DISP** (frmnam.rLdxll,offset-rl.r])	
	frmnam	form name
	offset	number controlling placement of form on screen
	Clears the portion of the screen specified as the “clear area” in the form description and displays a form. The display position can be offset from the original form description.	
DISPW	**FDV$DISPW** ([offset.rl.rj)	
	offset	number controlling placement of form on screen
	Clears the portion of the screen specified as the “clear area” in the form description and displays the form that is already loaded in the workspace. The display position can be offset from the original form description.	
DPCOM	**FDV$DPCOM** ([dpmode])	
	dpmode	value defining decimal point in signed-numeric fields
	Defines the comma, or redefines the period, as the decimal point in fields containing signed-numeric field-validation characters.	
DTERM	**FDV$DTERM** (tca.ml.da) or **FDV$DTERM** (tca.rtdxl)	
	tea	terminal control area
	Detaches a terminal from the Form Driver,and detaches any workspaces associated with the terminal.	
DWKSP	**FDV$DWKSP** (wksp.ml.da) or **FDV$DWKSP** (wksp.rt.dxl)	
	wksp	form workspace
	Detaches a form workspace from the list of workspaces associated with the current terminal.	
FIXJ3CREEN	**FDV$FIX_SCREEN** Repairs overwritten lines on the terminal screen.	
GET	**FDV$GET** (fldval.wt.dxl,fldtrm.wLr,fldnam.rt.dxl[,fldidx.rLr])	
	fidval	field value

Call	Procedure Parameter Notation	
	fldtrm	field terminator
	fldnam	field name
	fldidx	field index
	Positions the cursor in the initial cursor position of a specific modifiable field and waits for the operator to enter a value.	
GETAF	**FDV$GETAF** (fldvaLwkdxl41dtrm.wLr,fldnam,wtdxlf,fldidx.wl.r])	
	fidval	field value
	fldtrm	field terminator ending
	fldnam	field name ending
	fldidx	field index
	Positions the cursor in the current field in the form and waits for the operator to enter a value in any field.	
GETAL	**FDV$GETAL** ([fldval.wt.dxl,fldtrm.wl.r[,fldnam.rt.dxl[,fldidx.rl.rl]])	
	fidval	returned values of all fields in form
	fldtrm	field terminator starting
	fldnam	field name starting
	fldidx	field index
	Positions the cursor in the first modifiable field in a form unless otherwise specified in the **fldnam** argument and allows you to enter data in all modifiable, nonscrolled fields.	
GETDL	**FDV$GETDL** (value.wt.dxl41dtrm.wLr[,lme.rLr[,prompt.rt.dxl]])	
	value	contents of data line returned from Form Driver
	fldtrm	field terminator
	line	line number on which the operator's input is displayed
	prompt	data line text to serve as a prompt for the operator
	Gets a data line from a specified line on the screen.	
GETSC	**FDV$GETSC** (fldnam.rt.dxl,fidval.wtdxl[,fldtrm.wl.r])	
	fldnam	field name that identifies a scrolled area
	fidval	field values
	fldtrm	field terminator
	Positions the cursor within the current line in the scrolled area that contains the specified field and accepts input in modifiable fields within the line.	
ILTRM	**FDV$ILTRM** (trmmod.rhr)	
	trmmod	value for illegal terminator mode switch
	Specifies the action to take when an illegal field terminator is entered. An illegal field terminator can be rejected by the Form Driver or returned to the pro gram.	
LCHAN	**FDV$LCHAN** (channeLrLr)	
	channel	I/O channel number for form library

Call	Procedure Parameter Notation	
	Sets the channel for form library files associated with the current terminal. The Form Driver uses the specified channel for any LOPEN or LCLOS call processing.	
LCLOS	**FDV$LCLOS** Closes the form library associated with the current library channel for the current terminal.	
LEDOF	**FDV$LEDOF** (ledno.rLr)	
	ledno	terminal LED number
	Turns off the light-emitting diode (LED) on the VT100 keyboard.	
LEDON	**FDV$LEDON** (ledno.rLr)	
	ledno	terminal LED number
	Turns on the light-emitting diode (LED) on the VT100 keyboard.	
LOAD	**FDV$LOAD** (frmnam.rt.dxl)	
	frmnam	form name
	Loads a form description into a workspace without displaying the form on the screen.	
LOPEN	**FDV$LOPEN** (filspc.rt.dx1[,channeLrLr])	
	filspc	form library file specification
	channel	I/O channel number for form library
	Opens a form library and replaces the current library channel specification if the I/O channel number is supplied.	
NDISP	**FDV$NDISP** Marks current workspace as not displayed.	
PFT	**FDV$PFT** ([fldtrm.rl.r[,fldnam.rt.dxI[,fldvaLrt.dxl [,nfldnam.wLdxl[,nfldidx.wLr]]]]])	
	fldtrm	field terminator to be processed
	fldnam	field name that identifies a scrolled area
	fldval	field values to be displayed
	nflHnflm	current field name after call has been completed
	nfldidx	current field index after call has been completed
	Processes the field terminator and checks for valid terminator codes.	
PUT	**FDV$PUT** (fldval.rt.dxl3dnam.rt.dxlLfldidx.rlj*])	
	fldval	field value to be displayed
	fldnam	field name
	fldidx	field index
	Stores the value of the fldval argument and displays that value in the specified field.	
PUTAL	**FDV$PUTAL** (lfrmval.rt.dxl])	
	frmval	list of field values to be displayed

Call	Procedure Parameter Notation	
	Outputs values to all fields, stores the frmval argument values in the work-space for nonscrolled fields, and displays these values on the screen	
PUTD	**FDV$PUTD** (fldnam.rt.dxl[,fldidx.rLr])	
	fldnam	field name
	fididx	field index
	Outputs the default value to a specified field.	
PUTDA	**FDV$PUTDA** Outputs default values to all fields in the form and displays those values on the screen.	
PUTL	**FDV$PUTL** (text.rt*dxl[,line,rLr])	
	text	data line text
	line	line number for displayed data line
	Outputs data to the specified line on the screen. If the line number is zero, the data line is displayed on the last line of the screen.	
PUTSC	**FDV$PUTSC** (fldnam.rt.dxl[,fldval.rt.dxll)	
	fldnam	field name that identifies a scrolled area
	fldval	field value
	Outputs data to the current line of a scrolled area that contains the specified field name.	
READ	**FDV$READ** (frmnam.rt.dxl,mloc.ml,datmlocsiz.rl.r,frmsiz.wl.r) or **FDV$READ** (frmnam.rt.dxl^nloc.rt.dxl,mlocsiz.rLr,frmsiz.wl.r)	
	frmnam	form name
	mloc	form storage area
	mlocsiz	size of memory buffer that begins with **mloc**
	frmsiz	form size actually used
	Extracts a form from the current form library, stores it in a specified memory area, and adds the name of the form to the list of memory-resident forms.	
RET	**FDV$RET** (fldval.wtdxl,fldnam.rt.dx1l,fldidx.rl.r])	
	fldval	field value
	fldnam	field name
	fididx	field index
	Returns the value for a specified field stored in the workspace.	
RETAL	**FDV$RETAL** (frmval.wt.dxl)	
	frmval	concatenated values of all fields except those in scrolled areas
	Returns the values for all fields except those in scrolled areas stored in the workspace.	

Call	Procedure Parameter Notation	
RETCX	**FDV$RETCX** (atca.wa.r, awksp.wa.r, frmnam.wt.dxl, uarvaLwt.dxl, curpos.wl.r, fldtrm.wl.r, insovr.wl.r, hlpnum.wl.r)	
	atea	terminal control area address
	awksp	form workspace address
	frmnam	form name
	uarval	value of the associated text for this UAR
	curpos	cursor position within field
	fldtrm	returned field terminator
	insovr	Insert/Overstrike mode of field
	hlpnum	number of times HELP key pressed for current field
	Returns the current context of the Form Driver. You can issue this call in a UAR to determine the context in which the UAR is called.	
RETDI	**FDV$RETDI** (nmdidx.rl.r,nmdval.wt.dxl[,nmdnam.wt.dx1])	
	nmdidx	index of Named Data item
	nmdval	text of Named Data item
	nmdnam	name of Named Data item
	Returns the Named Data text that you specify by its index (rather than by its name).	
RETDN	**FDV$RETDN** (nmdnam.rt.dxl,mndval.wt.dxl[,nindidx.rl.r])	
	nmdnam	name of Named Data item
	nmdval	text of Named Data item
	nmdidx	index of Named Data item
	Returns the Named Data text that you specify by its name (rather than by its index).	
RETFL	**FDV$RETFL** (line.rl.r,value,wt.dxl,linlen.wl.r[,type.rl.r])	
	line	line number of form to be returned
	value	value of requested line
	linlen	length of character string returned in value parameter
	type	type of output line requested
	Returns the contents of the line that you specify with the **line** argument. This is one of the lines displayed by the RFRSH call. This call can also be used for loaded but undisplayed forms for report formatting.	
RETFN	**FDV$RETFN** (fldnam.wt.dxl[,fldidx.wl.r])	
	fldnam	field name
	fldidx	field index
	Returns the current field name and index from the current workspace. If the field is not indexed, the index value returned is zero.	
RETFO	**FDV$RETFO**	

Call	Procedure Parameter Notation	
	fldnum	field number
	fldnam	field name corresponding to **fldnum**
	fldidx	field index corresponding to **fldnum**
	Returns the name and index of the nth field in the form.	
RETLE	**FDV$RETLE** (fldlen.wl.r,fldnam.rt.dxl[,fldidx.rl.r))	
	fldlen	field length excluding field marker characters
	fldnam	field name
	fldidx	field index
	Returns the number of data characters in the specified field.	
RFRSH	**FDV$RFRSH** Refreshes the screen. The RFRSH operation is identical to that initiated by pressing the CTRL/R keys.	
SCR WIDTH	**FDV$SCR WIDTH** (width.lr.r)	
	width	80/132 column screen width
	Tells the Form Driver the current screen width.	
SIGOP	**FDV$SIGOP** Causes the application program to signal the operator.	
SPADA	**FDV$SPADA** (mode.rl.r)	
	mode	numeric/application keypad mode
	Sets the alternate keypad mode. Selecting 0 sets the alternate keypad to numeric mode; selecting 1 sets the keypad to application mode.	
SPOFF	**FDV$SPOFF** Turns supervisor-only mode off for the current terminal, allowing the operator to modify fields protected with the Supervisor Only attribute.	
SPON	**FDV$SPON** Turns supervisor-only mode on for the current terminal, treating fields protected with the Supervisor Only attribute as display-only fields.	
SSIGQ	**FDV$SSIGQ** (sigmd.rl.r)	
	sigmd	bell/reverse video signaling mode
	Sets signal mode for the current terminal. Audio mode (0) rings the terminal bell. Video mode (1) reverses the VT100/VT200 video image.	
SSRV	**FDV$SSRV** ([status.wl.r[,iostat.wl.r]])	
	status	general status reporting variable
	iostat	I/O status reporting variable
	Sets the addresses of the status reporting variables.	
STAT	**FDV$STAT** (status,wl.r[4ostat.wl.r])	
	status	general statuscode
	iostat	I/O statuscode

Call	Procedure Parameter Notation	
	Returns the statuscode for the last Form Driver call.	
STERM	**FDV$STEKM** (tca.ml.da) or **FDV$STERM** (tca.rt.dxl)	
	tea	terminal control area
	Sets current terminal and the workspace most recently associated with that terminal to the current workspace. The TCA must have been previously attached by the FDV$ATERM call.	
STIME	**FDV$STIME** (time.rl.r)	
	time	timeout period in seconds
	Specifies the number of seconds the Form Driver waits for operator response to a GET type call.	
SWKSP	**FDV$SWKSP** (wksp.ml.da) or **FDV$SWKSP** (wksp.rt.dxl)	
	wksp	form workspace
	Specifies the workspace that the Form Driver uses for the current workspace. The workspace must have been previously attached by the FDV$ATERM call.	
TCHAN	**FDV$TCHAN** (channel.rLr)	
	channel	physical I/O channel number for terminal
	Changes the terminal channel associated with the current TCA to the specified value.	
USER_REFRESH	**FDV$USERJREFRESH** ([rfr_address.ra-r])	
	rfr_address	user-supplied refresh routine
	Sets up a user-supplied refresh screen routine.	
WATT	**FDV$WAIT** ([fldtrm.wl.r])	
	fldtrm	field terminator code
	Causes the application program to wait until the operator presses a terminator key. This call allows the Form Driver to synchronize the application program to the operator’s pace.	