VMS Software

VSI OpenVMS

VSI DECforms
Programmer's Reference Manual

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

Software Version: DECforms Version 4.0

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

VSI DECforms Programmer's Reference Manual

VMS Software

Legal Notice

Table of Contents

Preface

This guide contains information about the Form Manager, the run-time component of DECforms software. It describes how to develop application programs to use with the Form Manager, how applications call Form Manager requests, and how the Form Manager processes these requests.

1. About VSI

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

2. Intended Audience

This manual is intended for application programmers and those interested in the operation of DECforms software at run time. You are expected to be familiar with a structured programming language, the Independent Form Description Language (IFDL), and the structure of a form in DECforms software.

3. Document Structure

This manual consists of the following chapters and appendixes:

Chapter 1	Gives an overview of run-time processing.
Chapter 2	Gives examples of calling requests from an application program, and explains how to use escape routines and the Form Manager Trace Utility.
Chapter 3	Describes how to use Oracle Trace software with DECforms applications.
Chapter 4	Describes how the Form Manager processes external requests.
Chapter 5	Describes the syntax of each of the OpenVMS external request calls and arguments.
Chapter 6	Describes the portable application programming interface for C and FORTRAN and how to use each of the external request calls.
Appendix A	Describes the state that causes each of the elementary conditions to be true.
Appendix B	Describes the receive control text items that the Form Manager can return.

4. Related Documents

See the online help, the online release notes, or the following documents for more information about DECforms:

- *VSI DECforms Installation Guide for OpenVMS Systems*—Describes how to install DECforms software on VAX and Alpha processors that are running the OpenVMS operating system.

- *VVSI DECforms Guide to Commands and Utilities*—Describes the DECforms forms commands and utilities.
- *VSI DECforms IFDL Reference Manual*—Describes the DECforms syntax information of the Independent Form Description Language (IFDL).
- *VSI DECforms Guide to Developing an Application*—Part I explains for the beginning DECforms programmer how to create a DECforms application, including both the form and the program. Part II contains additional guidelines and examples for more experienced DECforms programmers.
- *VSI DECforms Guide to Demonstration Forms and Applications*—Describes how to use various demonstration forms and applications. This guide is contained in online files named forms $demo_guide.txt and forms$demo_guide.ps in the FORMS$EXAMPLES directory.

  If you cannot find this document, ask your system manager to install it in the appropriate directory.

  For information about displaying these forms, see the appendix section of the *VSI DECforms Guide to Developing an Application*.
- *VSI DECforms Style Guide for Character-Cell Devices*—Describes how to develop user interfaces with a Motif style for DECforms applications for character-cell terminals.
- *VSI DECforms Guide to Converting FMS Applications*—Describes how to convert a VAX FMS or DEC FMS application to a DECforms application.

Also of interest to users of DECforms software is the *CODASYL Form Interface Management System Journal of Development* (see the Acknowledgment section).

5. OpenVMS Documentation

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

6. 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.

7. Conventions

The following conventions are 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.)

Convention	Meaning
. . .	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.
[|]	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 to Run-Time Processing

A DECforms application has the following components:

- The application program, which contains the data processing algorithms.
- The form, which contains the description of how to display information on the display device and what information to collect from the terminal operator.
- The display device, which displays information and optionally sends input from the operator to the form and program. The display device is not a component of the application; its declaration is a component of the form. DECforms software supports character-cell (VT) devices, and print output. For more information about device support, see Section 1.1.
- The Form Manager, which controls run-time processing in DECforms software.

The Form Manager displays information on the display device; accepts input from the terminal operator; and controls communication between the application program, the form, and the display device.

This chapter describes the use and operation of the Form Manager by introducing the following topics:

- Overview of DECforms device support
- How the Form Manager controls run-time communication
- How you can transfer data between the form and the application program
- How you control certain aspects of run-time processing
- How the Form Manager uses the activation list

1.1. Overview of DECforms Device Support

DECforms contains support for character-cell (VT) devices, and print output.

Print output is in DDIF (Digital Document Interchange Format). DDIF is the standard document format used by CDA, a component of HP Network Application Support (NAS) architecture that defines standards for compound documents and enables file interchange among all compliant applications.

1.1.1. Support for Character-Cell Terminals

Character-cell devices include VT devices such as the VT100- through VT500-series terminals, and terminal emulators that correctly emulate those VT terminals. DECforms supports these character-cell devices through the definition of viewports, panels, and responses in IFDL layouts having a device type of %VT100, %VT200, %VT300, %VT400, or %VT500. If you specify %VT100, you can run the layout on all VT devices.

Character-cell layouts support both input and output operations. They allow applications to display panels and data on the display device and also allow the user to enter and modify data through a

keyboard. You can specify particular user-defined actions in the character-cell layout for individual function keys on the keyboard.

To choose a character-cell layout as the user interface, the application must pass the device name of a character-cell device as the enable device parameter in the Forms Enable request. Often this name is SYS $INPUT, or SYS$OUTPUT. When the Form Manager must use a character-cell layout, it selects the layout in the form that best matches the characteristics of the specified device. This layout has a device type of %VTxxx and most closely matches the VT layout according to the type, width, height, and color capabilities of the device.

Character-cell layouts support viewports, panels, panel groups, icons, picture fields, text fields, text literals, polylines, and point literals.

A set of default function responses is defined by the Form Manager for the keyboard function keys. If you choose, you can override these definitions.

User input is validated against the field's input picture as each character is typed.

Character-cell layouts and their contents can be created and modified by the Form Development Environment and Panel Editor. Only on Alpha, Migration utilities can be used to aid the conversion of FMS form files to DECforms IFDL form files.

1.1.2. Support for PRINTER Devices

DECforms supports PRINTER devices through the definition of viewports, panels, and responses in IFDL layouts having a device type of %PRINTER. The primary use of PRINTER layouts is to generate high quality printable output.

The PRINTER device means an output file in DDIF format. You can convert a DDIF document you produced in DECforms to PostScript® format and print it on a PostScript printer. (To convert the file format, you must have the CDA converters supplied with DECwindows on your system.)

An application program depicts panels with the DISPLAY response step. This display does not actually appear on the terminal or workstation display device when you use a PRINTER layout. The DISPLAY response step instead creates output representing these panels. An application might print a set of invoice or purchase orders by using a PRINTER layout, or an application might simply generate a high-quality, fixed-format printable report by using this feature.

To use a %PRINTER device and choose a PRINTER layout, you must pass the file name as the enable device parameter in the ENABLE request. This layout must have a device type of %PRINTER.

To choose between multiple PRINTER layouts, you can specify an optional selection label within the IFDL file. An application can pass a selection label to the Form Manager to choose a specific layout (the layout with the matching selection label). In the absence of a selection label, the Form Manager chooses the first DDIF layout found. PRINTER layouts cannot be shared with VT layouts.

PRINTER layouts support viewports, panels, panel groups, picture fields, text fields, text literals, polylines, and point literals.

Function keys and function responses can be defined in a PRINTER layout, but are ignored at run-time. Because there is no operator interaction with a PRINTER layout, all such actions are ignored. However, the Form Manager processes a small subset of response steps (DISPLAY, LET, IF/THEN/ELSE...) in the external request processing phase.

You can create and modify PRINTER layouts by using any Editor.

If your form defines a PRINTER layout, you can create output representing the layout's panels and panel contents that is suitable for PostScript printing as follows:

1. In the application program, create an ENABLE request that passes the standard arguments:
   - Session identification—The session identification string created by the Form Manager
   - Device Name—The name of the display device to be used

     For DDIF printing, supply a file name (for example, *document_name.doc*).

     If you are using the DECforms portable Application Programming Interface (API), this argument is not required. (You can control the display device by using the FORMS$DEFAULT_DEVICE logical name; see step 2.)
   - Form File—The name of the form file
   - Form Name—The name of the form
   - ENABLE request options—Any request options
2. In the form, use external response declarations, such as ENABLE and SEND, to display each panel to be output, with its application data.

If you do not specify the display device in the ENABLE request, the operator must define a logical name at the system prompt before running the application by using the following command format:

```
$ DEFINE FORMS$DEFAULT_DEVICE document_name.doc
```

When the operator runs the application, DECforms creates a file in the format *document_name.doc* in the current directory.

The *VSI DECforms Guide to Developing an Application* provides an example of how to support quality printing in an IFDL form and in an application program.

1.1.3. Adding PRINTER Layouts Support to Character-Cell Applications

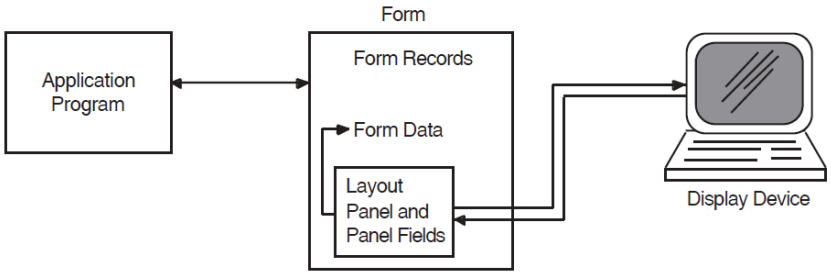
Adding PRINTER layouts to your application may require you to redesign part of your application. Because PRINTER layout devices are output-only, there is no operator input of data. Therefore, you may choose to define a different set of DECforms requests, or possibly a different application, that interacts with the PRINTER layout to produce high quality output. You can create and design the contents of PRINTER layouts by using any Editor.

1.2. Initiating and Controlling Run-Time Communication

During run time, the form, the display device, and the application program send data to and receive data from one another. The Form Manager controls this communication.

Figure 1.1 illustrates these interactions.

Figure 1.1. Run-Time Interaction



When the Form Manager initiates run-time processing, it loads a **form**, which is a data structure that controls a user interface to the application program. The form is specified as an argument to an ENABLE call and describes the data transfer and all the screen interactions that occur.

You have the following options for the location of forms at run-time:

Application Programming Interface (API)	Form Options at Run-time
OpenVMS API	Binary (disk-based) file Object-based (linked) form Shareable image file
Portable API	Binary (disk-based) file Object-based (linked) form

Once the form is loaded, a **layout** is selected. The layout is the appearance of the form on the display device and is defined within the form. After the layout is selected, the Form Manager creates a **session**, which is an interaction between the application program and the form that begins when the form is enabled.

Next, the display device is attached to the session, and the **form data** is initialized. Form data is a set of variables that are associated with a form and a session. Once the form data is initialized, it can be moved between the form, display device, and application program.

The following steps occur during the transfer of data between the form, display device, and application program:

1. The application program sends a **request** to the Form Manager using a subroutine call. The request contains information specifying the type of request, the name of a record, and points to an area of memory that contains record data.

2. When the request reaches the Form Manager, the data portion of the request exists as a **form record**. A form record is a structure that controls data transfer between the program and the form. The form record describes the structure of the record data. The Form Manager interprets the form record, which tells the Form Manager in what form data items to store the data received from the application program. Form records do not store form data or application program record data.

3. The Form Manager stores the data that it received from the application program in the appropriate **form data items**.

4. The Form Manager displays a **panel** on the operator display device. Panel fields on the panel display form data items. The Form Manager formats the panel and the data according to instructions that it read from the form before displaying the panel.

5. The Form Manager accepts operator input from the keyboard of the display device. The keyboard is used to enter data characters and functions. The Form Manager has a predefined set of functions that control operator interaction with the form.

   The form designer can modify these functions for a form definition. The Form Manager executes each function response and performs its actions when the function is entered by the operator. Instructions from the form tell the Form Manager in which form data items to store the data that it received from the display device.

6. The Form Manager stores the data that it received from the display device in the appropriate form data items.

7. The Form Manager returns data to the application program. DECforms stores the data in the area of memory pointed to by the request. The Form Manager uses the data structure defined by the form record. The application program receives a request from the Form Manager.

1.3. Transferring Data Between the Application and the Form

The external interface to the Form Manager consists of six function routines, which are called requests. Requests are used to enable forms, disable forms, transfer data between the form and the application program, or cancel outstanding requests.

When you enable a form, you establish an association between a form and a display device called a session. You also create an instance of the form data declared in the form. When this session is established, the Form Manager assigns a unique identifier to the session called the **session-id** that allows the Form Manager and the application program to identify the session.

When you disable a form, the Form Manager discards the association between the form and the display device, and the session-id that identifies the session. The Form Manager must enable a form before it can transfer data to or from that form. The Form Manager must disable a form before it can end the processing of an application. You cause the Form Manager to transfer data by calling one of six requests. The request you call depends on which type of data transfer you want the Form Manager to perform.

Table 1.1 describes the function of each request:

Table 1.1. DECforms Request Calls

Request	Function/Transfer Type
ENABLE	Enables a form.
DISABLE	Disables a form.
SEND	Sends data from the application program to the form.
RECEIVE	Sends data from the form to the application program.
TRANSCEIVE	Combines the actions of a SEND and a RECEIVE in a single request.

Request	Function/Transfer Type
CANCEL	Asynchronously stops all previously issued and currently active requests for a given session.

To call a request, write a routine call in your application program. You control what data is passed between the form and the application program by passing arguments in the routine call. For information on how to use routine calls in an application program, see Chapter 2.

When you use the OpenVMS API, DECforms requests will return an OpenVMS condition value to your application program. The condition value is an unsigned longword in which the low-order bit signifies success or failure of the request. The Form Manager returns this condition value to the return status variable in the application program.

When you use the portable API, the value returned is one of the FIMS status values defined in Table 6.4. In either the OpenVMS or portable API, the value indicates the most severe error that the Form Manager encountered while processing a request.

Requests can also return status to your application program by means of **control text**. Control text consists of special text codes that the Form Manager can issue when significant events occur during request processing. For more information about the format and return values of requests, see Chapter 5. For more information about receive control text descriptions, see Appendix B.

1.4. Using Responses to Control Run-Time Processing

The Form Manager responds to a request call by performing a set of predefined actions. For example, each time you call a request, the Form Manager verifies that the arguments in the call are valid. (For more information on these predefined actions, see Chapter 4.)

To control Form Manager processing, you define **responses** in your IFDL source file. In a response, you list a number of instructions to the Form Manager called **response steps**.

These response steps are actions that allow you to control the display of fields, icons, and panels, and change the order of form processing. When you process a request, you can also use response steps to call subroutines (escape routines), define help processing, reset the value of form data, and signal the operator. (Section 4.3.3 describes the effect of each response step.)

You can define one or more responses for each of the requests. There is no response processing in a cancel request.

The responses correspond to the requests as follows:

Response	Request
ENABLE RESPONSE	ENABLE
DISABLE RESPONSE	DISABLE
SEND RESPONSE	SEND
RECEIVE RESPONSE	RECEIVE
TRANSCEIVE RESPONSE	TRANSCEIVE

For more information on defining responses in your IFDL source file, see the *VSI DECforms IFDL Reference Manual*.

1.5. Using the Activation List

During the processing of most requests, the Form Manager solicits input from the display device. The Form Manager asks for input from the operator by allowing the operator to enter data into a particular field or panel on the display device. The form controls video attributes that can indicate to the operator which field or panel needs input.

When a field, panel, or icon is available for input, it is said to be *active* or *activated*. An item becomes activated when the Form Manager performs an ACTIVATE response step. More than one field, panel, or icon can be active at a given time; however, the operator can enter input only into the current active field, panel, or icon.

The Form Manager tracks the current active items by maintaining an ordered internal list called the **activation list.** At the start of a request the activation list is empty. Each item on the activation list is called an activation item. The activation item into which the operator is currently entering input is called the **current activation item**.

For purposes of discussion, activation items can be divided into four groups. The activation item groups determine what the activation item is associated with and what input the operator must enter to satisfy an activation item.

The four types of activation items are as follows:

- **Field activation items** are associated with panel fields (picture fields, text fields, and slider fields). The operator enters data and functions to satisfy a field activation item.
- **Icon activation items** (on character-cell layouts only) are associated with icons. An icon is an element, much like a panel field, that can contain graphics or text, but does not have associated form data items. The operator can only enter functions to satisfy an icon activation item.
- **Wait activation items** are associated with a panel or with the layout for the currently enabled form. The operator can only enter functions to satisfy a wait activation item.

Response steps, specified within responses, allow you to customize form processing. Responses are the actions that the Form Manager takes when a request is received. For example, when the Form Manager receives a request to enable a form, it executes the ENABLE RESPONSE. You then specify response steps within the ENABLE RESPONSE. For a complete description of response steps, see Chapter 4.

To complete the processing of an activation item, the Form Manager typically performs a **function response** that contains a POSITION response step. A function response is an action that occurs when an operator enters a function during input (for example, Next Screen or Tab).

The POSITION response steps in the function responses control the order of activation item processing and operator input. The POSITION response step specifies an activation item where the Form Manager should position next. For example, a typical POSITION response step is POSITION TO NEXT ITEM. This response step causes the Form Manager to begin accepting input for the next activation item on the activation list.

DECforms function responses can be one of two types: built-in or user-defined.

The built-in function responses are predefined by the DECforms software, and provide many standard intra-field editing capabilities, such as:

- Cursor movement

- Field erasure
- Moving to the next or previous field or panel
- Character deletion

The built-in function responses are bound by default to specific keys with FUNCTION declarations. The keys vary depending on the display device you are using. For more information on built-in functions and field editing, see the *VSI DECforms IFDL Reference Manual*.

If the function response is not built-in, you specify the function response and associate it with a key or key sequence by using a FUNCTION declaration in your IFDL file.

You create user-defined functions in your form. For examples of user-defined functions, see the *VSI DECforms Guide to Developing an Application*.

A RETURN response step specifies that **accept phase** (the time when operator input is allowed) should end. You typically terminate operator input by specifying a RETURN response step within a function response.

The following is an example of how the Form Manager controls operator input:

1. The Form Manager encounters an ACTIVATE response step that corresponds to a panel field named FIELD_ONE. It adds an activation item for that field to the activation list.
2. The Form Manager encounters an ACTIVATE response step that corresponds to a panel field named FIELD_TWO. It adds an activation item for that field to the activation list.
3. Because FIELD_ONE is the first item on the activation list, the Form Manager processes that activation item first when it begins to process the activation list.
4. The operator enters data into FIELD_ONE.
5. The operator uses the built-in NEXT ITEM function. (The built-in NEXT ITEM function response contains the POSITION TO NEXT ITEM response step.) Input into FIELD_ONE is complete.
6. The Form Manager performs the response step by making FIELD_TWO the current activation item.
7. The operator enters data into FIELD_TWO.
8. The operator uses the TRANSMIT built-in function. (TRANSMIT contains a RETURN response step.)
9. The Form Manager performs the response step by validating FIELD_ONE and FIELD_TWO, terminating accept phase, and returning control to the application program.

For more information on the activation list and how the Form Manager processes each activation item, see Section 4.4.

Chapter 2. Developing the Application Program

This chapter explains how to call DECforms requests. It also explains compiling, linking, and running an application program and how to use escape routines and the Form Manager trace facility.

The application program performs calculations, file operations, and record-keeping functions, and maintains the data being manipulated in the application (the data that the terminal operator sees and changes). To maintain the data, the application program must request that the Form Manager send data to the form or send data from the form to the application program by calling a DECforms request.

Occasionally, the form may initiate a data exchange itself by calling an **escape routine**. This process is called a **procedural escape**. A typical escape routine is a subroutine of the application program. The subroutine is called using a CALL response step. Data can be transferred between the form and the escape routine by means of arguments specified in the CALL response step.

Because of the complex run-time processing that occurs when a request or an escape routine is called, the Form Manager provides a trace facility. The trace facility records processing information that you can use to debug your application program and your form. The Form Manager also provides an event logging mechanism to trap any errors or unusual events during run time.

2.1. Using DECforms Requests

You call DECforms requests from an application program written in one of the programming languages that DECforms supports.

When you use the OpenVMS Application Programming Interface (API), DECforms conforms to the OpenVMS Calling Standard and supports the following languages:

- ADA
- BASIC
- BLISS
- C
- COBOL
- DIBOL
- FORTRAN
- Pascal
- PL/1

For more information about the OpenVMS API, see Chapter 5.

The portable API supports the C and FORTRAN binding on the OpenVMS operating system. It uses PASCAL calling conventions and FAR pointers. Any language that can construct argument lists that meet this standard can use this API. For more information about the portable API, see Chapter 6.

When you use a request in an application program, you must format the request as you would any other function call written in the programming language that you use. For example, in FORTRAN you should designate that an argument is omitted from a DECforms request by including a null entry in place of the argument.

This section explains how to use the requests from a FORTRAN application program. The examples are from the advanced FORTRAN sample application program, written using the portable API. This sample application maintains a checking account and a savings account. The program is located in the FORMS $EXAMPLES directory, which is an option you can choose to install during the DECforms installation procedure.

To view this advanced sample application program, print the following file:

FORMS$EXAMPLES:FORMS$CHECKING_FORTRAN.FOR

Note

The portable API uses header files– formsdef.h (for C) and formsdef.f (for FORTRAN).

These header files contain definitions of data types that DECforms requires. You need to declare data using the correct data types.

For example, the declaration for a form name called form_name in the IFDL source form is as follows in the C application program:

```
Forms_Form_Object form_name;
```

The form used for the advanced sample checking application is in FORMS$EXAMPLES:FORMS $CHECKING_FORM.IFDL.

When the advanced application program is running, the operator can do the following:

- Write checks on the account.
- Deposit money into the account.
- Withdraw cash from the account.
- Transfer money from the checking account to the savings account.
- Transfer money from the savings account to the checking account.
- Review and change account information (for example, the address of the account's owner).
- Review the history of transactions made on the account.

For more information about the sample programs, see the *VSI DECforms Guide to Developing an Application*.

2.1.1. Enabling Requests

Before the Form Manager can display information, accept input, or control communication, it must load a form into memory, select a layout, attach a display device, allocate an instance of form data, and create

a session-identification string. Because the Form Manager performs these tasks during the processing of an ENABLE request, the first request that you call from your application program must be the ENABLE request.

If you are using multiple forms, you do not need to place all enable calls before any other request calls. You need to ensure only that the ENABLE request, FORMS$ENABLE (OpenVMS API) or forms_enable (portable API), is the first request made for a given display device and form pair.

Example 2.1 contains a portion of the advanced FORTRAN sample application program written for the portable API that calls the ENABLE request.

Example 2.1. Enabling a Form from the Advanced FORTRAN Sample Program

```
        IMPLICIT NONE
        INCLUDE 'forms_checking_common.f'                          ❶
        INTEGER sample_checking_account
        EXTERNAL sample_checking_account
C
C Define data and form file names
C
        CHARACTER*200  form_file_name
        CHARACTER*200  sample_data_name
C Option list.  Used to pass special options to forms requests (calls)
        RECORD /forms_request_options/ enable_request_options(3)

C Print startup message on console

        PRINT *,
     1 'FORTRAN DECforms Sample Checking Account Application starting.'

C Build the data and form file names

        CALL forms_checking_getdir( form_file_name, sample_data_name)

C Set up printing to go to the operator's scratch directory. This involves
C passing a request_options parameter in the enable request.

        enable_request_options(1).option = forms_c_opt_print
        enable_request_options(1).print_file_name = %LOC(print_file_name)
        enable_request_options(1).print_file_name_length =
 LEN(print_file_name)

        enable_request_options(2).option = forms_c_opt_form
        enable_request_options(2).form_object =
 %LOC(sample_checking_account)

        enable_request_options(3).option = forms_c_opt_end

C Initialize the DECforms form & check for errors



                                                                  ❷
   forms_status = forms_enable_for( session_id,         ! session id string
 1                                  device_name_string, ! Device name
 2                                  form_file_name,     ! Name of form file
 3                                  form_name_string,   ! Name of form
```

```
4                                   enable_request_options)
                                                     ! Request options
list

       CALL check_forms_status( forms_status )                    ❸

   .
   .
   .
```

❶ The INCLUDE statement includes the forms_checking_common file. This file contains the declaration of common areas to be used in this application program.

❷ The ENABLE request call passes the following arguments:

- session_id

  Variable that contains the 16-character session-identification string created by the Form Manager. The session-identification string is returned to the application program in this argument upon completion of this ENABLE request.

- device_name_string

  Display device to be used for this session.

- form_file_name

  Name of the form file to be used for this session.

- form_name_string

  Name of the form to be used for this session, as specified in the IFDL file.

- enable_request_options

  A list specifying one or more request options used to control the request environment. For more information, see Chapter 6.

In response to this request, the Form Manager loads the form with the specified form file name into memory and selects a layout in the form that is appropriate for the device (if a layout exists). The Form Manager also attaches the display device. After the Form Manager attaches the display device, it creates a session-identification number that it stores in the session_id variable.

For more details on processing that the Form Manager performs for the ENABLE request, see Chapter 4.

❸ The CALL statement passes the value returned in the forms_status variable to a subroutine that verifies that the ENABLE request was completed. The check_forms_status subroutine specifies that an error message be displayed and that the application program stop executing if the forms_status variable contains a value indicating an error.

Use the ENABLE request when you want to load a new form into memory, select a new layout, and attach a new display device. Each time that you call the ENABLE request, the Form Manager creates a new session, which is identified by a session-identification string. You can call the ENABLE request as often as you need to create new sessions.

The Form Manager creates a session-identification string for each session that you create. You must pass a session-identification string in all SEND, RECEIVE, TRANSCEIVE, DISABLE, and CANCEL

requests to tell the Form Manager which session to use while processing the request. Even if you create only one session, you must pass a session-identification string. Sessions are independent of one another; the only way to pass information between sessions is through the application program.

For more information on the ENABLE request and its arguments, see Chapter 5 and Chapter 6.

2.1.2. Moving Data from a Form to a Program

Use the RECEIVE request to retrieve data stored in form data items and move the data into the record fields of an application program record.

Records defined in your application program control how data is transferred to and from the program. You pass the application program record as a record message argument to a request call. The record message argument is a pointer to an area of memory that stores the fields in the application program record.

When the Form Manager transfers data to the application program during the processing of a request, it copies the data into the area of memory pointed to by the record message. These values are kept in the application program record until the program itself or the processing of another request changes them.

Example 2.2 contains a portion of a subroutine written for the portable API. This subroutine gets new account information from the operator and moves that new information into the account record stored in the application program. The new information is moved during the processing of the RECEIVE request.

Example 2.2. RECEIVE Request from the Advanced FORTRAN Sample Program

```
    .
    .
    .

C Get new information for account record.  If termination was quit,
C then the operator might have changed a few things that the quit is
C supposed to ignore, so send the original account record back to the
C form and return to menu processing.

    .
    .
    .

        record_data.data_length = account_size         ! for pre-V5 FORTRAN
C-V5 -> record_data.data_length = SIZEOF(account_temp) ! for VAX FORTRAN V5
        record_data.data_record = %LOC(account_temp)
        record_data.shadow_record = 0
        record_data.shadow_length = 0


C Set up to receive control text back from the form

        request_options(1).option = forms_c_opt_receive_control
        request_options(1).receive_control_text_count=
 %LOC(receive_ctl_txt_ct)
        request_options(1).receive_control_text=
 %LOC(receive_ctl_txt_string)

        request_options(2).option = forms_c_opt_end
```

```
C Get the record from the form

     forms_status = forms_receive_for( session_id,     ! session id   ❶
     1                                 'account',      ! form record
     2                                 record_data,    ! info from the form
     3                                 request_options)!request option list

        CALL check_forms_status( forms_status )                        ❷

    .
    .
    .
```

❶ This statement calls the RECEIVE request. Table 2.1 explains the arguments passed in this request.

Table 2.1. Request Arguments

Argument	Explanation
session_id	Variable containing the session-identification string (from an ENABLE request) that identifies the session to be used during this request.
’account ’	Name of the form record that contains record fields corresponding to form data items in the form. Each form data item that the application program can receive data from must correspond to a field in the form record. This correspondence allows the form record name to indicate to the Form Manager which form data items from which to move data.
record_data	Information received from the form.
request_options	A list specifying one or more request options used to control the request environment. For more information, see Chapter 6.

❷ The CALL statement calls a subroutine that determines whether the RECEIVE request was completed. The check_forms_status subroutine specifies that an error message be displayed and that the application program stop executing if the forms_status variable contains a value that indicates an error.

To move the values from form data items to fields in the ACCOUNT application program record (because the form activates fields in the RECEIVE RESPONSE), the Form Manager must solicit input from the operator. It does so by displaying the set of panel fields corresponding to the form data items associated with the data in the application program record.

The operator enters input to the panel fields, and the Form Manager stores that input in form data items. The Form Manager moves the values stored in the form data items to the fields in the application program record.

The application program record, the form record, the form data items, and the panel fields correspond as follows:

- The application program record is logically equivalent to the form record. This means that the application program record has the same number of fields as the form record, and that each field

in the application program record has the same data type, length, and dimension as the fields in the form record. Any alignment and padding requirements must be the same for both the application program record and the form record.

- The names of fields in the form record usually correspond to the names of form data items. The form associates each form data item with its corresponding field name in the form record.
- The name of each form data item that requires operator input corresponds to the name of a panel field. The form declares each panel field that has the same name as a form data item that requires input.

This correspondence allows the Form Manager to determine which panel fields need input, which form data items store that input, and into which application program record fields it moves this input during the processing of a RECEIVE request.

This name correspondence is the most common way to specify the transfer of form data to form records, but it is not the only way to transfer information from the form to your program. Another way to specify data transfer is to use the DATA TRANSFER clause. For more information, see the *VSI DECforms IFDL Reference Manual*.

If errors occur during the processing of this RECEIVE request, the Form Manager stores **receive control text items** in the variable for the receive_control_text request option. For more information about receive control text, see Section 4.7.

Use the RECEIVE request when you need information from the form and, optionally, from the operator in your application program. Not all RECEIVE requests solicit input from the operator, but soliciting input from the operator is part of the default processing that the Form Manager performs for the RECEIVE request. Section 4.3 describes the default processing.

The statements in the sample application program shown in Example 2.2 are part of a subroutine that is called when the operator chooses to look at and modify data stored in the account record. The account record is maintained in the application program.

This record stores the following data:

- Full name of the owner of the account
- Address of the owner of the account
- Account owner's work and home telephone numbers
- Date that the account was established
- Password that the operator must enter to change data in the record

Because the account record is maintained in the application program, the program must receive new values entered by the operator. Therefore, the program calls the RECEIVE request when the operator needs to enter new data into this record. This call ensures that the program receives the new data directly after it is entered.

As another example of using the RECEIVE request, suppose that the purpose of your application program is to maintain a database of employees who work in a particular department. In this application program, you include at least two RECEIVE requests:

- One RECEIVE request causes the Form Manager to solicit input from the operator and pass new information about employees whose records exist in the employee database. The statements before

and after this RECEIVE request open an existing employee record and write the new information that was received from the form into that record. The application program then replaces the updated record in the database.

- The other RECEIVE request causes the Form Manager to solicit input from the operator and pass information about new employees who do not have a record in the employee database. The statements before and after this RECEIVE request create the new employee record and write information about a new employee into that record. These statements create a place for that record and store the record in the database.

For more information on the RECEIVE request and its arguments, see Chapter 5 and Chapter 6.

2.1.3. Sending Data from a Program to a Form

Use the SEND request to send data to the form. This request causes the Form Manager to move data from an application program record to form data items.

Example 2.3 contains statements that update account information stored in the form using the SEND request. This example is written for the portable API.

Example 2.3. SEND Request from the Advanced FORTRAN Sample Program

```
C Update the balances, next check number, and room_in_reg flag in the form.
C If there's no room for more entries in the register, then
C the room_in_reg flag is sent as zero(false), else true(all 1's in
 Fortran).

        IMPLICIT NONE
        INCLUDE 'forms_checking_common.f'

        RECORD /update/ update                                          ❶

        update.checking_balance = checking_balance                      ❷
        update.savings_balance  = savings_balance
        update.check_number     = last_check_num
        update.room_in_reg      = register.number_entries_used .LT.
 reg_size

C Initialize the descriptor with UPDATE info

   .
   .
   .
        record_data.data_length = update_size    ! for pre-V5 FORTRAN
C-V5->  record_data.data_length = SIZEOF(update) ! for VAX FORTRAN V5
        record_data.data_record = %LOC( update )
        record_data.shadow_record = 0
        record_data.shadow_length = 0

C Send the update record

        forms_status = forms_send_for( session_id,      ! session id   ❸
     1                                 'update',        ! form record
     2                                 record_data,     ! info sent to form
     3                                 0 )              ! request options
 list
```

```
CALL check_forms_status( forms_status )                    ❹

RETURN
END
```

❶ The RECORD declaration names the ’update ’ form record as a data structure.

❷ The assignment statements assign values to fields in the ’update ’ form record. The values that are assigned to these fields were determined in another part of this program. To see the subroutine that determines the contents of these variables, refer to the full listing of this application program. See Section 2.1 for the location of the advanced sample program.

❸ This statement calls the SEND request. The following list explains the arguments that are unique to this request. For explanations of the others, refer to Table 2.1.

- ’update ’

  The name of the form record that controls where the data passed in the SEND request is stored. The data is stored in the form data items that are named the same as the fields of the update form record.

- record_data

  The information that is sent to the form.

In response to this request call, the Form Manager updates the values of the form data items associated with the field's ’update ’ record. It performs this update by moving the contents of the following record fields into the following form data items:

Record Fields	**Form Data Items**
update.checking_balance	checking_balance
update.savings_balance	savings_balance
update.check_number	last_check_num
update.room_in_reg	register.number_entries_used

Note that the associated form record fields and form data items do not have to have the same names.

For more information on the description of the DATA TRANSFER clause, see *VSI DECforms IFDL Reference Manual*. For more information on how the Form Manager processes SEND requests, see Chapter 4.

❹ The CALL statement calls a subroutine that determines if the SEND request was completed. The check_forms_status subroutine specifies that an error message be displayed and that the application program stop executing if the forms_status variable contains a value that indicates an error.

Use the SEND request when your application program needs to change the values stored in form data items. The statements in Example 2.3 are used in a subroutine in the sample application program. This subroutine is called from other subroutines in the sample application program when the operator changes the account balance.

For example, assume that the application program had previously called a RECEIVE request. During the processing of the RECEIVE request, the operator writes a check on the account, and the amount of that check and the check number are returned to the application program. The application program then calculates a new account balance, assigns a new value to the highest check number, and verifies that the history register it maintains has room for more transactions.

If the history record has room to store the history record of more transactions, the application program sets the variable room_in_reg to 1. If the history record is full, the application program sets the room_in_reg variable to 0, and the operator cannot enter any more transactions on the account. (In this sample application program, the history record is limited to storing information about 30 transactions. Normally, the history record would not be limited in this way.)

Once the application program has performed these tasks, it calls the subroutine shown in Example 2.3. The SEND request passes the new balance, the highest check number, and the room_in_reg value to the form data items that store them. The contents of these form data items control what transactions the operator can perform. Therefore, it is important that these form data items be updated promptly after each operator action that modifies the account. If this is not done, the operator might be able to write a check for more than the current account balance.

As another example, suppose that the purpose of your application program is to calculate subtotal and total amounts for an invoice. Once these amounts have been calculated, you use the SEND request to pass the subtotal and total to form data items, which are then displayed on a panel. In this type of application program, using the SEND request immediately after the subtotal and total amounts are calculated ensures that incorrect values are not inadvertently displayed on the display device.

For more information on the SEND request and its arguments, see Chapter 5 and Chapter 6.

2.1.4. Asynchronous SEND Requests

In general, DECforms requests cannot be made from asynchronous system trap (AST) routines because much of DECforms request processing occurs at AST level, and normal request processing is synchronous with respect to the caller. Form Manager requests from AST level might produce deadlock situations within the process running a DECforms application.

However, it is often useful to communicate with a form from an AST routine. For example, an application might have a timer AST routine fire periodically to display time-critical data on the screen. DECforms software provides a mechanism by which the SEND request can be used at AST level to send information to the form. Specifically, when an AST routine issues a SEND request, the request executes asynchronously with respect to the calling routine. In addition, only a severely restricted subset of request processing is performed.

The asynchronous SEND requests perform only the data distribution phase of request processing and then the request is terminated. If the data being sent to the form is currently displayed on a panel, the panel fields are updated normally with the new values. However, because no response processing or accept phase processing is performed, the panel must be displayed by a previous synchronous request.

Because SEND requests at AST level execute asynchronously with respect to the AST routine, the programmer must take care to provide proper synchronization at request completion. DECforms can provide AST notification of request completion, using the FORMS$K_ASTADR and FORMS $K_ASTPRM item codes in the request-options item list argument in the OpenVMS API.

Hewlett-Packard Company strongly recommends that you use this feature when using asynchronous SEND requests. For more information on the FORMS$K_ASTADR and FORMS$K_ASTPRM item codes, see Chapter 5. For more information about setting up AST routines, see the OpenVMS System Services manual.

In the portable API, the FORMS$K_ASTADR and FORMS$K_ASTPRM item codes are referred to as completion routines. You can set up the mechanism by using the completion routine structure in the request option definition.

For examples of using asynchronous send requests, see the following:

FORMS$EXAMPLES:FORMS$DEMO_TIMER.EXE
FORMS$EXAMPLES:FORMS$DEMO_TIMER_AST.COB
FORMS$EXAMPLES:FORMS$DEMO_TIMER_AST_ROUTINE.COB
FORMS$EXAMPLES:FORMS$DEMO_TIMER_CHECK_STATUS.COB
FORMS$EXAMPLES:FORMS$DEMO_TIMER_COMP_ROUTINE.COB
FORMS$EXAMPLES:FORMS$DEMO_TIMER_FORM.IFDL
FORMS$EXAMPLES:FORMS$DEMO_TIMER_SET_TIMER.COB

2.1.5. Transceiving Data

Use the TRANSCEIVE request to send data to the form from the application program and to receive data from the form in the application program.

Example 2.4 contains an example of using the TRANSCEIVE request in the portable API. For this TRANSCEIVE request, the application program is sending and receiving two fields in register_record.

Example 2.4. TRANSCEIVE Request from the Advanced FORTRAN Sample Program

```
C
        IMPLICIT NONE
        INCLUDE 'forms_checking_common.f'

C Option list.  Used to pass special options to forms requests (calls)

        RECORD /forms_request_options/ request_options(2)

C Initialize the descriptor with REGISTER info

   .
   .
   .
        record_data.data_length = register_record_size  ! for pre-V5
 FORTRAN ❶
 C-V5-> record_data.data_length = SIZEOF(register)      ! for VAX FORTRAN
 V5
        record_data.data_record =   %LOC(register)
        record_data.shadow_record = 0
        record_data.shadow_length = 0



C Set up to receive control text back from the form

        request_options(1).option = forms_c_opt_receive_control
        request_options(1).receive_control_text_count=
%LOC(receive_ctl_txt_ct)
        request_options(1).receive_control_text=
%LOC(receive_ctl_txt_string)

        request_options(2).option = forms_c_opt_end



C Transceive the record (send it and ask to get it back)
```

```
forms_status = forms_transceive_for(session_id,! session id      ❷
1                                   'register_record',  ! send record name
2                                   record_data,        ! the send record
3                                   'register_record',  ! Receive record
                                                        ! name
4                                   record_data,        ! the recv record
5                                   request_options)    ! request options
                                                        ! list

CALL check_forms_status( forms_status )                          ❸

END
```

❶ The values for the size and location of the register record are stored in the record named register_record in a different subroutine in this program.

❷ The next statement calls the TRANSCEIVE request. The following table explains the arguments passed in this request.

Argument	Explanation
session_id	Variable containing the session-identification string (from an ENABLE request) that identifies the session to be used during this request.
’register_record ’	Name of the form record that controls where the data passed in this request is stored. The data is stored in the form data items that have the same names as the fields of the register_record form record.
record_data	Send record.
’register_record ’	Name of the form record that controls where data is to be returned to the application program.
record_data	Receive record.
request_options	A list specifying one or more request options used to control the request environment. For more information, see Chapter 6.

In response to the TRANSCEIVE request, the Form Manager sends the value for register_record to the form by moving the data in the register_record application program record fields into the register_record form data items.

The Form Manager then solicits input from the operator and returns that input to the application program. The Form Manager solicits input into panel fields that correspond to the register_record form data items. The Form Manager stores the input in these form data items and moves the new values in the form data items to the register_record application program record fields.

The send-record and the receive-record parameter descriptors can refer to different records; you can send data to one record stored in the form, and receive data from a different record stored in the form.

For more information on Form Manager processing of the TRANSCEIVE request, see Chapter 4.

❸ The CALL statement calls a subroutine that determines whether the TRANSCEIVE request was completed. The check_forms_status subroutine specifies that an error message be displayed and that the application program stop executing if forms_status contains a value that indicates an error.

Note

The subroutine shown in this example is a simplified version of a subroutine that transceives a record contained in the sample application program. For information on how the TRANSCEIVE request is actually used in the sample application program, see the documentation of the sample applications in the *VSI DECforms Guide to Developing an Application*.

Use the TRANSCEIVE request any time information you send to the form could generate new information that you need in your application program. For example, suppose your application program calculates and stores the values needed to create a balance sheet from your accounting records. This program might contain a TRANSCEIVE request within a loop.

In the TRANSCEIVE request, you send the numbers that were calculated in the application program to the form. These numbers are then displayed on the display device in a completed balance sheet. The operator looks at the balance sheet to see whether it is correct.

If the balance sheet is correct, the operator enters a value in a field that indicates the balance sheet is complete. When this occurs, the Form Manager returns the same values to the application program that it just received from the application program. These values are stored for future reference.

However, if the balance sheet is incorrect, the operator enters a value in a field that indicates the balance sheet is incorrect. The Form Manager then asks the operator for input into each field that affects the totals on the balance sheet.

Once the operator has entered all the changed information, the new data is returned to the application program. The application program recalculates the amounts to be displayed in the balance sheet and returns them to the display by calling the TRANSCEIVE request again. This process continues until the balance sheet is correct.

For more information on the TRANSCEIVE request and its arguments, see Chapter 5 and Chapter 6.

2.1.6. Disabling Requests

Call the DISABLE request to terminate the session.

Example 2.5 contains part of a FORTRAN application program that calls the DISABLE request.

Example 2.5. DISABLE Request from the Advanced FORTRAN Sample Program

```
C Clean up, Print ending message on console, leave.

   IMPLICIT NONE

   INTEGER status
   INCLUDE 'forms_checking_common.f'

   forms_status = forms_disable_for( session_id , 0 )                ❶

   PRINT *, 'FORTRAN DECforms Sample Checking Account Application ending.'
   STOP
   END
```

❶ The application program calls the DISABLE request to detach the display device and unload the form. The session_id argument passed in this request tells the Form Manager which display device to detach and which form to unload. The 0 parameter indicates that there are no request options.

For more information on Form Manager processing of the DISABLE request, see Chapter 4.

2.1.7. Canceling Requests

In some situations, you may want to cancel the processing of the current DECforms requests immediately. Use the CANCEL request to stop processing of the current request.

Example 2.6 contains part of a FORTRAN application program that calls the CANCEL request.

Example 2.6. CANCEL Request from the Advanced FORTRAN Sample Program

```
forms_status = forms_cancel_for( session_id,0)    ❶
CALL check_forms_status( forms_status )           ❷
END
```

❶ The application program calls the CANCEL request to cancel the active DECforms request. The session_id argument passed in this request tells the Form Manager for which session to cancel the request.

❷ Once the request processing is complete, the application program passes the value returned in the forms_status variable to the check_forms_status subroutine. This subroutine specifies that an error message be displayed and that the application program stop executing if forms_status contains a value that indicates an error.

For more information on Form Manager processing of the CANCEL request, see Chapter 4.

2.2. Compiling, Linking, and Running the Application

Once you have written your application program, you can compile, link, and run it. For all the following examples, references to the sample program demonstrate commands using the OpenVMS API, and references to the checking program demonstrate commands using the portable API.

2.2.1. Compiling an Application

To compile your application program in the OpenVMS or portable API, enter the command that invokes the compiler for your programming language. For example, to compile the FORTRAN sample application programs, enter one of the following FORTRAN commands:

```
$ FORTRAN FORMS$SAMPLE_PROGRAM_FORTRAN  (OpenVMS API)
$ FORTRAN FORMS$CHECKING_FORTRAN (portable API)
```

The command in this example creates an object file that can be input to the OpenVMS Linker. In this example, the object files are named FORMS$SAMPLE_PROGRAM_FORTRAN.OBJ (OpenVMS API) and FORMS$CHECKING_FORTRAN.OBJ (portable API). For information about the command that invokes your compiler, see the documentation for your programming language.

The FORTRAN compiler picks up the library file in your working directory, and the C compiler picks up the library file in SYS$INCLUDE.

To compile the C sample application program in either the OpenVMS or portable API, enter either of the following C commands:

```
$ CC FORMS$SAMPLE_PROGRAM_C (OpenVMS API)
$ CC FORMS$CHECKING_C (portable API)
```

2.2.2. Linking an Application

After you compile your program, enter the LINK command to invoke the OpenVMS Linker. On the command line, specify the file name of the object file that your compiler created. For example, to link the FORTRAN sample application program that uses the OpenVMS API, enter the following command:

```
$ LINK FORMS$SAMPLE_PROGRAM_FORTRAN
```

The command in this example creates an executable image that you can execute with the RUN command. In this example, the executable image is named FORMS $SAMPLE_PROGRAM_FORTRAN.EXE. For more information on the LINK command, see the *OpenVMS Linker Reference Manual*.

To link the advanced FORTRAN sample application program that uses the portable API, enter the following command:

```
$ LINK FORMS$CHECKING_FORTRAN.OBJ, SYS$INPUT/OPTION -
_$ SYS$SHARE:FORMS$PORTABLE_API/SHARE Ctrl/Z
```

If you plan to link escape routines with your application program, you should link the escape routines, your application program, and the form object file on the same command line. Section 2.3 describes using escape routines.

All DECforms entry points for the new portable bindings are in the shareable image, SYS $SHARE:FORMS$PORTABLE_API.EXE. The DECforms run-time system remains in SYS $SHARE:FORMS$MANAGER.EXE.

When you use a linked form in the portable API, you must specify the /PORTABLE_API qualifier in the FORMS EXTRACT OBJECT command to generate the global form name symbol in the form object.

To obtain the form object file for the OpenVMS API, enter the following command:

```
$ FORMS EXTRACT OBJECT FORMS$SAMPLE_FORM.FORM
```

To obtain the form object file for the portable API, enter the following command:

```
$ FORMS EXTRACT OBJECT /PORTABLE_API FORMS$CHECKING_FORM.FORM
```

2.2.3. Running an Application

After you compile and link your application program, you can enter the RUN command to execute it. To execute the program, include the file name of the executable image that the Linker creates on the RUN command line. For example, to run the FORTRAN sample application program, enter one of the following commands:

```
$ RUN FORMS$SAMPLE_PROGRAM_FORTRAN (OpenVMS API)
$ RUN FORMS$CHECKING_FORTRAN (portable API)
```

For more information on the RUN command, see the *OpenVMS DCL Dictionary*.

2.3. Writing and Calling Escape Routines

An escape routine is an application program subroutine that is called during the processing of a request. In an escape routine, you specify actions that you want performed during request processing. For example, you might write an escape routine that performs a database lookup or a file operation.

You can write an escape routine in any of the programming languages that DECforms software supports. You can call a request from an escape routine. However, you cannot call a DISABLE request that terminates the session from which the escape routine was called.

You must call escape routines from responses in your IFDL source file using the CALL response step. The arguments that you pass in the CALL response step name the escape routine and pass values that the escape routine needs. When the Form Manager encounters the CALL response step, it finds and transfers control to the escape routine. For more information about the CALL response step, see Chapter 4.

If you want more than one application to have access to a set of escape routines, store the escape routines in a separate file. To access the escape routines from several applications, compile the escape routines and either link them in a shareable image (in the OpenVMS API only) or link them with your application program.

Section 2.3.4 describes the effect of storing escape routines in a shareable image. Section 2.3.1 and Section 2.3.2 describe linking escape routines with your application program. You can also store some escape routines in a shareable image and link others with your application program. Section 2.3.6 lists the steps you follow to do this.

2.3.1. Creating a Form Object

To link an escape routine with your application program or to enable a linked form, you must create a form object. A form object contains a list of the escape routines that are called from the form file you are using. You create the form object using the Extract Object Utility.

To use the Extract Object Utility to create a form object, enter a command similar to the following. Use the /PORTABLE_API qualifier for portable API applications.

```
$ FORMS EXTRACT OBJECT filename.FORM /OUTPUT=form_obj
```

In this case, the resulting object file is named form_obj.obj. If you do not specify a destination object file using the /OUTPUT qualifier, the Extract Object Utility creates a file with the same name as the input file name and a file type of `.obj`. For more information on the FORMS EXTRACT OBJECT command, see the *VSI DECforms Guide to Commands and Utilities*.

The Extract Object Utility stores the names of all the escape routines that you call from your form in a form object module. You must link this object with your application program. When you link the form object, the Linker assigns addresses to those escape routines that are linked directly with your application program. If the Linker encounters the names of escape routines that are not linked directly with your application program, it assigns a value of zero to those names.

When the Form Manager prepares to transfer control to an escape routine, it searches for the name of the escape routine in FORMS$AR_FORM_TABLE. If the Form Manager finds an address assigned to the name, it transfers control to the subroutine that starts at that address. However, in the portable API, if the Form Manager finds a value of zero assigned in the table, DECforms returns an error indicating that it could not find the escape routine. In the OpenVMS API, it attempts to activate a shareable image to find the escape routine.

Once the Form Manager finds the escape routine, it transfers control to the first instruction in that escape routine.

Note

When an escape routine calls another form request on the same session, the built-in form data item PARENTREQUESTID must be passed to the escape routine and subsequently passed as a parameter

to the forms request. Failing to pass PARENTREQUESTID as a parameter could result in a deadlock situation. Because multiple requests are active on the same session, the Form Manager does not schedule the request for execution of the second form request until the first request completes unless PARENTREQUESTID is passed for the second request.

By passing PARENTREQUESTID to the second request, the Form Manager determines that the second request should be scheduled and allowed to execute before the first request completes.

An escape routine which calls another form request on a different session does not need to pass PARENTREQESTID as the two sessions are independent.

2.3.2. Linking Escape Routines Directly with an OpenVMS API Program

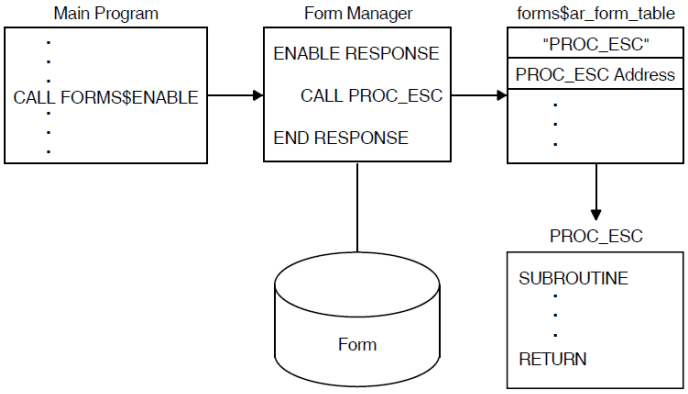
To link escape routines directly with your application program, enter the LINK command. For example, to link the FORTRAN sample application program and an escape routine using the OpenVMS API, you might enter the following command:

```
$ LINK FORMS$CHECKING_FORTRAN, ESC_UNIT, FORM_OBJECT
```

In this example, the application program, FORMS$CHECKING_FORTRAN.OBJ, is linked with one escape routine, ESC_UNIT.OBJ, and one form object. The form global symbol, FORMS $AR_FORM_TABLE, contains the address of ESC_UNIT after the Linker finishes processing this command.

Figure 2.1 illustrates how the Form Manager transfers control to an escape routine that has been directly linked with the application program.

Figure 2.1. Control Transfer to a Directly Linked Escape Routine



The following steps describe the process outlined in Figure 2.1.

1. The application program calls the Form Manager with the ENABLE request (FORMS$ENABLE).

2. The Form Manager executes the ENABLE response. The response contains a CALL response step, which includes an escape routine called proc_esc.

3. The Form Manager searches for the proc_esc routine, which it locates in the FORMS $AR_FORM_TABLE table. The Form Manager then transfers control to the proc_esc routine.

2.3.3. Linking Escape Routines Directly with a Portable API Program

To link escape routines directly with a program in the portable API, you need to do the following in your portable program:

1. Declare a global form name as an external integer in FORTRAN or use the predefined type, Forms_Form_Object, in the C language.
2. Specify the form name and a request option list in the ENABLE call.
3. Set up a form object option by using the global form name in the enable request option list.

For detailed examples on setting up a form object option in the enable request option list, see Section 6.2 and Section 6.3.

To link escape routines with your portable program and form object, enter the following commands:

```
$ FORMS TRANSLATE MYFORM.IFDL

$ FORMS EXTRACT OBJ /PORTABLE_API MYFORM.FORM /OBJ=MYFORM.OBJ

$ LINK MAIN_PROGRAM, ESC_UNIT, MYFORM.OBJ, SYS$INPUT /OPTION -

_$ SYS$SHARE:FORMS$PORTABLE_API /SHARE Ctrl/Z
```

2.3.4. Linking Escape Routines in a Shareable Image

You can link escape routines in a shareable image *only* if you are using the OpenVMS API. For information on creating a shareable image, see the *OpenVMS Linker Utility Manual*.

To resolve escape routine references in your form, you must create a form object file using the forms Extract Object Utility and link it with your escape routines and application program.

After you create a shareable image, you must either define the FORMS$IMAGE logical name or specify the name of the image in the request-options argument of the ENABLE call. If you use the logical name, it must be defined as the name of the shareable image that contains the escape routine you want to use. For example, the following command defines the FORMS$IMAGE logical name to point to a shareable image name SHAREABLE_IMAGE:

```
$ DEFINE FORMS$IMAGE DISK1$:[SMITH]SHAREABLE_IMAGE
```

All routines in the shareable image that you invoke as escape routines must be universal symbols in the shareable image. Escape routines are located at run time with the LIB$FIND_IMAGE_SYMBOL routine. For more information on LIB$FIND_IMAGE_SYMBOL, see the OpenVMS Run-Time Library documentation.

If you are using more than one shareable image, the FORMS$IMAGE logical name can be a search list. You cannot specify a node name in this logical name definition; however, you can specify a device, directory, and file type. If you omit the device and directory from the logical name definition, the image activator looks for the file in SYS$SHARE. If you omit the file type, the image activator looks for a file with the .EXE type.

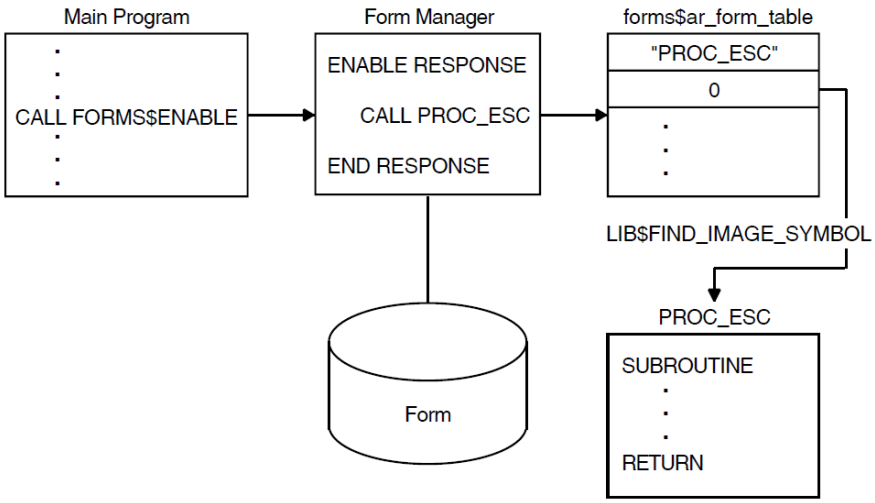
If the FORMS$K_IMAGE item code was specified in the request-options argument of the ENABLE request, the FORMS$IMAGE logical name is not used.

When you specify the item code, the buffer address should contain the name of the shareable image containing the escape routines. You can specify the FORMS$K_IMAGE item code up to eight times, each referring to a different shareable image. If you specify more than one shareable image, the Form Manager begins searching with the first shareable image in the request-options list and terminates processing when it finds the procedural escape or it tries all the shareable images in the item list.

If you did not specify a FORMS$K_IMAGE item code in the enable response, or the Form Manager cannot translate FORMS$IMAGE when it attempts to transfer control to an escape routine stored in a shareable image, it returns an error to your application program. If the Form Manager cannot find the shareable image, it also returns an error to your application program. In either case, the request that caused the escape routine to be called terminates processing.

Figure 2.2 illustrates how the Form Manager transfers control to an escape routine that has been stored in a shareable image.

Figure 2.2. Control Transfer to an Escape Routine in a Shareable Image



The following steps describe the process outlined in Figure 2.2:

1. The application program calls the Form Manager with the ENABLE request (FORMS$ENABLE).

2. The Form Manager executes the ENABLE response. The response contains a CALL response step, which includes the proc_esc escape routine.

3. The Form Manager searches for the proc_esc escape routine in the FORMS$AR_FORM_TABLE table. The Form Manager finds a 0 for the address of the escape routine.

4. Using the RTL routine LIB$FIND_IMAGE_SYMBOL, the Form Manager locates the escape routine and transfers control to the proc_esc escape routine.

2.3.5. Building Applications with Shared Forms or Shared Procedural Escape Routines on OpenVMS Alpha

When you build applications that use shared images that contain either forms or procedural escape routines on OpenVMS Alpha, you must specify the following Linker option when building these shared images:

```
symbol_vector=(FORMS$AR_FORM_TABLE=DATA)
```

For example, to build the DECforms sample application form as a shared image, issue the following command:

```
$ LINK/SHARE/EXE=FORMS$SAMPLE_FORM FORMS$SAMPLE_FORM, SYS$INPUT/OPT -
_$ SYMBOL_VECTOR=(FORMS$AR_FORM_TABLE=DATA)
```

The shared image can then be linked with the application program, allowing the form to be accessed through this image.

In addition, if the procedural escape routines are contained in a shared library that does not contain an .obj file that was created by issuing the FORMS EXTRACT OBJECT or the FORMS EXTRACT OBJECT/NOFORM_LOAD command, each shared procedural escape routine entry point must be declared as well.

For example, if the procedural escape routines (get_deposit, get_check, and print_panel) are in a separate .obj file sample_peus.obj:

```
$  LINK/SHARE/EXE=SAMPLE_PEUS, SAMPLE_PEUS, SYS$INPUT/OPT -
_
$ SYMBOL_VECTOR=BOLD
_$ BOLD
```

2.3.6. Combining the Direct Link and Shareable Image Methods

To link escape routines directly with your program and also use escape routines stored in shareable images use the following procedure:

1. Create the escape routines and the responses that call them.
2. Create the shareable image containing the escape routines.
3. Define the FORMS$IMAGE logical name as the name of the shareable image, or specify the FORMS$K_IMAGE item code in the ENABLE call.
4. Create the form object.
5. Link the application program, the object module, and any escape routines *not* stored in the shareable image.
6. Run your application program.

Note

You can use a combination of the direct link and shareable image methods to link escape routines to your application program only with the OpenVMS API.

2.3.7. Enhancements to Escape Routine Debugging

Debugging DECforms escape routines that exist in shareable images is difficult because the Form Manager dynamically activates the images in the escape routines using the RTL LIB $FIND_IMAGE_SYMBOL routine. You cannot set breakpoints in escape routines until the Form

Manager activates the image. You must signal the SS$_DEBUG condition in routines that are to be debugged as procedural escapes.

To make it less difficult to debug escape routines, you can specify a logical name, FORMS $DEBUG_ESCAPE_ROUTINES, at session-enable time. If you specify the logical name FORMS $DEBUG_ESCAPE_ROUTINES as true, the Form Manager invokes the Debugger by signaling SS$DEBUG the first time an escape routine is called for a given DECforms session. FORMS $DEBUG_ESCAPE_ROUTINES is true when it is defined as a character string value that begins with any of the following characters: 1, T, t, Y, or y.

To invoke the Debugger, you must link the executable image that calls the Form Manager with traceback.

Once the Form Manager invokes the Debugger, you must enter the appropriate SET IMAGE, SET MODULE, and SET BREAK debug commands to set up a debugging environment. A GO command is also required. The SS$DEBUG condition is signaled as close as possible to the actual CALLG instruction that invokes the escape routine.

2.4. Using the Trace Facility

The Trace facility, supplied in the Form Manager, logs form processing information at run time to help you debug your application program and your form. This facility is useful because a great deal of form processing occurs each time you call a request from your application program.

Note

Because of the size of the trace files and the overhead incurred in writing the trace files, tracing is not recommended in a production environment. You should be sure to turn tracing *off* before creating your final run-time environment.

An alternative to tracing, the event log, is described in Section 2.5. Because event logging does not require much overhead, you might want to keep it on in your final run-time environment.

The Trace facility writes the following information to a file each time a request is called:

- Which request was called
- Responses and response steps performed
- Activation items processed
- Input functions accepted
- Names of panel fields or groups for which entry, exit, or validation responses are performed
- Form Manager logical names translated
- Records transferred to and from the form
- Record message fields transferred to and from the form

2.4.1. Controlling Tracing

Table 2.2 describes the options you have for turning on the tracing facility.

Table 2.2. Methods for Turning On the Trace Facility

Time of Action	API	Action
Before running the application	OpenVMS API Portable API	Define the logical name FORMS $TRACE to T,t,Y,y, or 1. Define the logical name FORMS $TRACE_FILE to the name you want for the trace file.
Per request during execution of program	OpenVMS API	Set the request option item code to FORMS$K_TRACE. For details on using request options with the OpenVMS API, see Chapter 5.
		Set the request option item code to FORMS$K_TRACEFILE. For details on using request options with the OpenVMS API, see Chapter 5.
	Portable API	Set the request option to forms_c_opt_trace, and set trace.file_name and trace.file_name_length. For details on using request options with the portable API, see Chapter 6.

Trace Files

The first time tracing is turned on, the Form Manager opens a trace file specified by either the FORMS $K_TRACEFILE item code or the FORMS$TRACE_FILE logical name on the OpenVMS API, or the trace.file_name and trace.file_name_length variable on the portable API. If you specify the item code or option, the logical name is ignored. If you want the trace messages to be written to your display device, you can specify the trace file to be TT:.

If you do not specify the item code, the FORMS$TRACE_FILE logical name, the FORMS_TRACE_FILE file name, or if trace.filename is undefined when the trace facility is turned on, the trace information is written to a file in your current directory.

The file name is the same as the file name of the form file. If the file name is not specified, the form name specified in the form file is used. The file type of the file is .TRACE. For example, if your .FORM file is named FORM_ONE.FORM, the default trace file name is FORM_ONE.TRACE.

The trace file is defined as a sequential file with variable-length records.

Using the FORMS$TRACE Logical Name

The trace facility sends information to the trace file when the FORMS$TRACE logical name is defined as being on (set to T, t, Y, y, or 1). The trace facility stops writing information to the trace file when FORMS$TRACE is defined as being off. FORMS$TRACE is defined as being off when it is defined as a character string value that begins with any character *except* T, t, Y, y, or 1, or when it is not defined.

You can check the validity of the FORMS$TRACE logical name only on an ENABLE request. If you did not specify the FORMS$K_TRACE item code or the trace.flag field in the request-options parameter, the FORMS$TRACE logical name is translated and tracing is turned on if necessary.

Tracing Multiple Sessions

When you turn on the trace logical name for a process running multiple sessions, the DECforms software opens a trace file with a unique version number to record the progress of each session separately.

Using Request Options for Tracing

You can start and stop tracing by specifying the FORMS$K_TRACE item code in the OpenVMS API, or the trace.flag field in the portable API in the request-options parameter for each request. A zero in the buffer address terminates tracing and a nonzero value starts tracing. Tracing remains on or off until you specify a request that changes it.

2.4.2. Enabling and Disabling Tracing

Although it is convenient to enable tracing, there may be occasions where you wish to turn tracing on and off. For example, if you have a very large application, you may wish to trace only one of the requests through the entire application.

You can turn the trace facility on and off by specifying the FORMS$K_TRACE item code in OpenVMS API and the trace.flag field in the portable API in the request-options argument of each request call. Example 2.7 shows an example of a portable API C program that turns tracing on and off.

Example 2.7. Portable API C Program that Uses Tracing

```
#include <stdio.h>
#include <formsdef.h>

#define TRACE_FILE_NAME "testdate.trc"

/* Declare the data structure. */



typedef
  struct _record_message_type {
      char    text_field[15];
      long    num_field;
  } Record_Message_Type;

 /* Forms Request Parameters */

  Forms_Session_Id          SessionId;
  Forms_Record_Data         Send_Message_Desc;
  Forms_Record_Data         Receive_Message_Desc;
  Forms_Request_Options     forms_request_options[10];

  Record_Message_Type    Send_Message;
  Record_Message_Type    Receive_Message;

  Forms_Status Status;

  Forms_Control_Text Receive_Ctl_Text;
  Forms_Count  Receive_Ctl_Text_Count = 5;
```

```
  Forms_Control_Text Send_Ctl_Text;
  Forms_Count  Send_Ctl_Text_Count = 5;
  Forms_Value  Timeout;



main(int argc, char **argv)
{
  int i;


/* Set up the request options to establish the trace file name. The file*/
/* must be set on enable, but the flag can be passed on any request.   */

  forms_request_options[0].option = forms_c_opt_trace;
  forms_request_options[0].trace.file_name = TRACE_FILE_NAME;
  forms_request_options[0].trace.file_name_length =
 strlen(TRACE_FILE_NAME);
  forms_request_options[0].trace.flag = 0; /* The flag means "no
 tracing".*/



/* Setup Enable Options List for the Enable call. */

  i = 1;
  /* receive control text info           */
  forms_request_options[i].option = forms_c_opt_receive_control;
  forms_request_options[i].receive_control.text = &Receive_Ctl_Text;
  forms_request_options[i].receive_control.text_count =
 &Receive_Ctl_Text_Count;
  i++;
  /* send control text info              */
  forms_request_options[i].option = forms_c_opt_send_control;
  forms_request_options[i].send_control.text       = Send_Ctl_Text;
  forms_request_options[i].send_control.text_count = Send_Ctl_Text_Count;
  i++;

  forms_request_options[i].option = forms_c_opt_end;



  /* Issue the Forms Enable call to test a bad Enable call */

  Status = forms_enable(SessionId,            /* Form Session Id ptr*/
                        0,                    /* Terminal Device    */
                        "sample",             /* Form File name     */
                        0,                    /* Form name          */
                        forms_request_options); /* Options list       */
    if (Status != forms_s_normal)
 return (1);



  /* To enable tracing on this request: */

   forms_request_options[0].option = forms_c_opt_trace;
   forms_request_options[0].trace.flag = 1;
```

```
   forms_request_options[1].option = forms_c_opt_end;

  /* Initialize the send record structure. */
  Send_Message_Desc.data_record   = &Send_Message;
  Send_Message_Desc.data_length   = sizeof(Send_Message);
  Send_Message_Desc.shadow_record = NULL;
  Send_Message_Desc.shadow_length = 0;



  Status = forms_send(SessionId,
                      "SAMPLE_RECORD",
                      &Send_Message_Desc,
                      forms_request_options);
  if (Status != forms_s_normal)
      {
      Status = forms_disable(SessionId, 0);
      return (1);
      }



  /* To disable tracing: */

   forms_request_options[0].option = forms_c_opt_trace;
   forms_request_options[0].trace.flag = 0;

   forms_request_options[1].option = forms_c_opt_end;



  /* Initialize the receive structure. */
  Receive_Message_Desc.data_record    = &Receive_Message;
  Receive_Message_Desc.data_length    = sizeof(Receive_Message);
  Receive_Message_Desc.shadow_record  = NULL;
  Receive_Message_Desc.shadow_length  = 0;

  Status = forms_receive(SessionId,
                         "SAMPLE_RECORD",
                         &Receive_Message_Desc,
                         forms_request_options);


  if (Status != forms_s_normal)
      {
      Status = forms_disable(SessionId, 0);
      return (0);
      }

  Status = forms_disable(SessionId,              /* Form Session Id ptr*/
                         forms_request_options); /* Options list       */

  return (0);
}
```

Example 2.8 shows an example of an ENABLE request that turns tracing on in the OpenVMS API.

Example 2.8. Using Tracing with the FORMS$ENABLE Call

```
*+
* Sample COBOL syntax that shows how to use the
* request options item list to enable tracing.
*-
IDENTIFICATION DIVISION.
PROGRAM-ID.    EMPLOYEE.
DATA DIVISION.
WORKING-STORAGE SECTION.


*+
* Pull in the FORMS definitions file.
*-


COPY "sys$share:forms$cob_definitions".                                ❶
*+
* Information that is transferred between this program and DECforms
*-
01  session_id                      PIC X(16) GLOBAL.
01  file_name                       PIC X(9)  VALUE "EMPLOYEE_PANEL".
01  device_name                     PIC X(9)  VALUE "SYS$INPUT".
01  forms_status                    PIC S9(9) COMP GLOBAL.


*+
* Request options item list.
*-
01 request_options.                                                    ❷
    03 item_1.
        05 buff_len_1               PIC S9(4) COMP.
        05 item_code_1              PIC S9(4) COMP.
        05 buff_addr_1              PIC S9(9) COMP.
        05 ret_len_1                PIC S9(9) COMP.
    03 item_2.
        05 buff_len_2               PIC S9(4) COMP.
        05 item_code_2              PIC S9(4) COMP.
        05 buff_addr_2              PIC S9(9) COMP.
        05 ret_len_2                PIC S9(9) COMP.
    03 end_of_item_list             PIC S9(9) COMP VALUE 0.


*+
* Values for item list.                                                ❸
*-
01 tracefile                        PIC X(9) VALUE "TRACEFILE".        ❹
01 tracefile_addr                   POINTER VALUE IS REFERENCE tracefile.
01 tracefile_len                    PIC S9(4) COMP VALUE 9.

01 trace_off                        PIC S9(9) COMP VALUE 0.            ❺
01 trace_on                         PIC S9(9) COMP VALUE 1.
01 trace_len                        PIC S9(9) COMP VALUE 4.            ❻
PROCEDURE DIVISION.
0.
```

```
*+                                                                      ❼
* Load first item in request options item list.
* This item causes tracing to be turned on.
*-
    move forms$k_trace to item_code_1.
    move trace_len to buff_len_1.
    move trace_on to buff_addr_1.
    move binary_zero to ret_len_1.


*+
* Load second item in request options item list.                        ❽
* This item specifies the name of the trace file to
* be created.
*-
    move forms$k_tracefile to item_code_2.
    move tracefile_len to buff_len_2.
    move tracefile_addr to buff_addr_2.
    move binary_zero to ret_len_2.


*+
* Enable the form and create the DECforms session.                      ❾
*-
     CALL "forms$enable" USING OMITTED
                               BY DESCRIPTOR device_name
                               BY DESCRIPTOR session_id
                               BY DESCRIPTOR file_name
                               OMITTED
                               OMITTED
                               OMITTED
                               OMITTED
                               OMITTED
                               OMITTED
                               OMITTED
                               BY REFERENCE request_options

                     GIVING forms_status.


*+
* Test the completion status.
*-
    IF forms_status IS FAILURE THEN
        CALL "LIB$SIGNAL" USING BY VALUE forms_status
        STOP RUN
    END-IF.
END PROGRAM EMPLOYEE.
```

❶ The forms$cob_definitions file contains the declaration of the FORMS$ENABLE routine name, and the FORMS$K_TRACE and FORMS$K_TRACEFILE symbols.

❷ The request_options declaration creates a two-dimensional array containing two item lists.

❸ The next set of declarations assigns constant values to be used in the item lists.

❹ The tracefile constant contains the name of the file that is to contain trace information. The tracefile_addr variable contains the address of the tracefile constant. The tracefile_len variable contains the length of the tracefile constant.

❺ The trace_on and trace_off constants control whether tracing is on or off. To turn tracing on, pass the trace_on constant. To turn tracing off, pass the trace_off constant.
❻ The trace_len constant contains the length of the trace_on or trace_off constants. The value 4 indicates that the constants are each a VAX longword.
❼ The assignment statements move values into the first item list, which is the item_1 list. The item_1 list controls whether tracing is on or off. In this case, tracing is turned on because the trace_on constant is assigned to the buff_addr_1 variable.
❽ The second set of assignment statements moves values into the item_2 item list. This list controls where trace information is written.
❾ The FORMS$ENABLE call invokes the ENABLE request. The FORMS$ENABLE call passes a number of empty parameters, the file_name and device_name constants, and the session_id variable. It also passes the request_options item list, containing the tracing information.

2.4.3. Exception Conditions During Tracing

If the Form Manager encounters an exception condition during tracing, it issues a warning message and returns a receive control text item describing the error condition to the application program. If an exception condition occurs, tracing is turned off for the current session.

The following conditions cause the Form Manager to turn off tracing:

- The trace facility cannot open the trace file.
- The trace facility cannot write information to the trace file.
- The trace facility cannot close the trace file.

2.4.4. Tracing Command Procedure

Example 2.9 shows a sample DCL command procedure that invokes the trace facility and specifies the file in which the trace facility writes trace information. The FORTRAN advanced sample application program is run in this example.

Example 2.9. Sample DCL Command Procedure Invoking the Trace Facility

```
$! Turn on tracing
$ DEFINE FORMS$TRACE "T"
$!
$! Define a subdirectory for the output filespec
$ DEFINE FORMS$TRACE_FILE USER:[SMITH.TEST]MY_TEST.TRACE
$!
$ DEFINE/USER SYS$INPUT SYS$COMMAND
$!Run the application program
$ RUN FORMS$CHECKING_FORTRAN
$!
$!Type out the trace file when through processing
$ TYPE USER:[SMITH.TEST]MY_TEST.TRACE
```

2.4.5. Capturing Additional Tracing Information

If you want to include additional tracing information about data movement, define the OpenVMS logical name FORMS$TRACE_CONVERSIONS to T. The resulting trace file will be much larger than a normal trace file.

2.5. Using the Event Log

Using the trace facility at development time is useful, but it might be inappropriate for a production environment. During development time, you may require detailed information about the actions of the Run-Time Manager. In a production environment, you might not need to trace every action of the Run-Time Manager, although it may be important to trap any errors or unusual events.

The Event Log file provides a broader scope in capturing error data than the trace file. The Event Log file writes only information pertinent to the logged event. The Event Log file reports all errors within the Form Manager, whereas the trace file reports only errors pertaining to request processing.

You enable Form Manager Event Log files by setting the logical name FORMS$LOG_EVENTS to 1. If an error occurs in your form sessions, a log file named FORMS$EVENT_LOG.LOG appears in your working directory. You can change the name of the log file by setting the logical name FORMS $LOG_FILE_NAME. You can set these logical names prior to or during program execution. You can purge, delete, or rename the log file while the program is running. DECforms creates a new version of the log file each time it encounters an error.

This information may be helpful should you submit the error as a problem report to HP. Example 2.10 outlines the information contained in an Event Log file.

Example 2.10. Event Log File

```
DECforms error logged on *date* *time*

Process ID: *process_id*
     * Process Name: *process_name*
     * Image Name:   *program_name*


*error_message_text*
      .
      .
      .


   + Signal Vector [0] : *error_vector_data*
   + Signal Vector [1] : xxxxxxx

   .
   .
   .
   .


   + Mechanism Vector [0] : *error_vector_data*
   + Mechanism Vector [1] : xxxxxxx
   + Mechanism Vector [2] : xxxxxxx
   + Mechanism Vector [3] : xxxxxxx


     MGR_form: Terminal Name: *terminal_name*

    + RCB: Address: xxxxxxx
    + RCB: Control Bits: xxxxxxx
    + RCB: Thread Id: xxxxxxx
```

```
  + RCB: Additional Control Bits: xxxxxxx
  + SCB: Address: xxxxxxx
  + SCB: Control Bits: xxxxxxx
  + SCB: Scheduling Bits xxxxxxx
  + SCB: Current Request: xxxxxxx
  + SCB: VM Zone Cache: xxxxxxx, Verify status: xxxxxxx
  + SCB: Zone Cache User: xxxxxxx
  + SCB: Error Routine: xxxxxxxx; Error Param: xxxxxxx


    Param Block: Request Type: *request_type*
  + Param Block: Request Flags: xxxxxxx
    Param Block: Session ID: *session_id*


 + VM Zone: wmg_a_display_zone: xxxxxxx, verify status: xxxxxxx
 + VM Zone: wmg_a_windows_zone: xxxxxxx. verify status: xxxxxxx
 + VM Zone: wmg_a_rectangle_zone: xxxxxxx, verify status: xxxxxxx
 + VM Zone: wmg_a_buffers_zone: xxxxxxx, verify status: xxxxxxx
 + VM Zone: wmg_a_frame_zone: xxxxxxx, verify status: xxxxxxx


  *+ *raw_stack_dump_data*
  .
  .
  .
  .
*request_type* values:


    forms_enable      = 1
    forms_send        = 2
    forms_receive     = 3
    forms_transceive  = 4
    forms_disable     = 5
    forms_cancel      = 6
    forms_send(async) = 7


    Legend:
             *     = OpenVMS only
             +     = DECforms Internal information
             *...* = User applicable data
```

Chapter 3. Using Oracle Trace Software with DECforms Applications

DECforms has many predefined duration and point **events** that occur during run time. (For a complete list of events, see Table 3.1.) An event can have a start and an end or it can simply occur. Oracle Trace allows events within DECforms to be defined and DECforms data items to be associated with each event. These data items can be standard resource utilization items or items specific to applications using DECforms.

Oracle Trace records several different pieces of information, called **items**, for each event. Items can be information about the event itself, such as the name of the event or in what procedure the event is occurring. Items can also include process statistics and performance information, such as the working set size at the time of the event. Section 3.1.2 contains descriptions of DECforms events and items.

To use Oracle Trace commands directly from VSI Command Language (DCL), preface them with the keyword COLLECT. For example:

```
$ COLLECT SHOW VERSION
Oracle Trace Version V2.2
$
```

You can also enter the Oracle Trace command environment by entering the COLLECT command with no arguments at the DCL prompt. Oracle Trace software prompts you for commands until you return to DCL command level with the EXIT command.

For example:

```
$ COLLECT
Trace> SHOW VERSION
Oracle Trace Version V2.2
Trace> EXIT
$
```

3.1. How to Collect Event Data

To collect event data with Oracle Trace software, you must first create a facility selection and then schedule data collection using that selection.

3.1.1. Creating a Selection

Oracle Trace software can collect event data from a number of layered products and applications. These other products are referred to as **facilities**, and each has a facility definition registered in the Oracle Trace administration database. You can use the SHOW DEFINITION command to list the facilities installed on your system.

The following example shows the facility definitions.

```
$ COLLECT SHOW DEFINITION /FORMAT=NAMES_ONLY
14-APR-1997 14:36       Facility Definition Information
```

```
Page 1
Names Only Report                                          Oracle Trace
 V2.2

 Facility:             Version:    Creation Date:      Class:
 --------------------  ----------  ------------------  ------------------
 FORMS                 V2.2        14-APR-1997 04:21   ALL
                                                       MONITORING
                                                       PERFORMANCE
(D)
                       V2.1         1-MAR-1997 23:36   ALL
                                                       MONITORING
                                                       PERFORMANCE
(D)

$
```

Oracle Trace software can collect data from all the registered facilities or just the ones in which you are interested.

To specify a subset of the available facilities, you use the CREATE SELECTION command to create a **facility selection**, which consists of the following items:

- Selection name
- List of facilities for which to collect data
- Class of data to collect for each facility

The basic format for the CREATE SELECTION command is as follows:

```
$ CREATE SELECTION selection_name /FACILITY=(facility_name[,...])
```

The following example defines the selection MY_SELECTION to collect the default data for DECforms:

```
$ COLLECT CREATE SELECTION MY_SELECTION /FACILITY=FORMS
```

To define a more detailed selection, you should use the OPTIONS qualifier, which takes a file name as an argument. The options file lists each facility and the collection class you want to use. Each facility is described on a separate line. If you specify OPTIONS but do not include a file name, Oracle Trace software prompts you for the options.

The format of the facility description in the options file is as follows:

```
$ FACILITY facility-name [/VERSION="version-code"] [/CLASS=class-name]
```

The name of the facility for which to collect data. A text string identifying the version of the facility. The string must be enclosed in quotation marks (" "). The class of data that you want collected for the facility.

The following example creates the facility selection SELECT_ALL to collect all the possible data for DECforms software and only the performance-related data from MY_FACILITY:

```
$ COLLECT CREATE SELECTION SELECT_ALL /OPTIONS
Options> FACILITY FORMS /CLASS=ALL
Options> FACILITY MY_FACILITY/CLASS=PERFORMANCE
Options> Ctrl/Z
$
```

3.1.2. Describing Events and Items

Each time a predefined event occurs in a DECforms application, Oracle Trace software records the event items in a data file. You can use this collected data to identify information for a variety of uses, such as how frequently an event occurs, in what order events occur, or how long an event takes to complete.

Note

Oracle Trace software collects all occurrences of each event in your chosen collection class. You cannot choose individual events to record. However, you can create reports based on specified events, by using the REPORT command. For information on Oracle Trace reporting, see Section 3.2.

Table 3.1 describes the *events* that can occur in your DECforms applications.

Table 3.1. Events

Event	Description	Type of Event
SESSION	Tracks the length of a DECforms session.	Duration
ENABLE	Tracks the length of an ENABLE request.	Duration
SEND	Tracks the length of a SEND request.	Duration
RECEIVE	Tracks the length of a RECEIVE request.	Duration
TRANSCEIVE	Tracks the length of a TRANSCEIVE request.	Duration
DISABLE	Tracks the length of a DISABLE request.	Duration
CANCEL	Logs the occurrence of a CANCEL request.	Point
DISTRIBUTION	Tracks the length of time needed to perform data distribution for a particular record.	Duration
COLLECTION	Tracks the length of time needed to perform data collection for a particular record.	Duration
CALL_RESPONSE	Tracks the length of time needed to process a CALL response step in IFDL.	Duration
ACCEPT_PHASE [1]	Tracks the length of time from the end of the accept phase to the start of the next accept phase (the processing time of a transaction when not accepting user input).	Duration
PANEL_DURATION	Tracks the length of time spent accepting input within an individual panel.	Duration

[1]The ACCEPT_PHASE event can be interpreted as the response time of a transaction.

Table 3.2 describes the *items*, related to the events in the previous table, that are specific to DECforms applications.

Table 3.2. Data Items

Item	Description	Data type
session id	Name of the DECforms session.	fixed_ascic
form name	Name of the DECforms form.	fixed_ascic
send record name	Name of the SEND record.	fixed_ascic
receive record name	Name of the RECEIVE record.	fixed_ascic
escape routine name	Name of the escape routine.	fixed_ascic
panel name	Name of the panel.	fixed_ascic

The ALL class is composed of the full set of events and items available for collection from DECforms. Table 3.3 lists the events and items that make up the ALL collection class.

Table 3.3. Events and Items Available in the ALL Class

Event	Items
SESSION	session id, form name
ENABLE	session id, form name
SEND	session id, form name, send record name
RECEIVE	session id, form name, receive record name
TRANSCEIVE	session id, form name, send record name, receive record name
DISABLE	session id, form name
CANCEL	session id, form name
DISTRIBUTION	session id, form name, send record name
COLLECTION	session id, form name, receive record name
CALL_RESPONSE	session id, form name, escape routine name
ACCEPT_PHASE [1]	session id, form name
PANEL_DURATION	session id, form name, panel name

[1]The ACCEPT_PHASE event can be interpreted as the response time of a transaction.

3.1.3. Scheduling Data Collection

You must schedule data collection on your system for Oracle Trace software to begin collecting information about DECforms software.

The scheduling criteria include:

- Output file for the collected data
- Start and end times (or alternately, the duration)
- Which facility selection to use
- Whether to collect from your entire cluster or just the local node

Oracle Trace software does not allow you to schedule data collections that overlap or run simultaneously; although you can schedule many data collections on a node, only one collection can be active on a node at any time.

You can schedule data collection either locally or cluster wide, by using the [NO]CLUSTER qualifier. By default, the SCHEDULE COLLECTION command schedules data collection on every node in the cluster. To schedule data collection on a subset of the cluster, you must log in to each node that you want data collection to occur on and schedule local data collection on that node by specifying NOCLUSTER. On a standalone system, the CLUSTER qualifier is ignored.

The following example schedules the collection MY_COLLECTION to begin at 11:00 and end at 12:00 on the current day. The collection uses the facility selection SELECT_ALL and runs on the local node. Oracle Trace software stores the collected data in the file MY_DATA.DAT in your default device and directory.

```
$ COLLECT SCHEDULE COLLECTION MY_COLLECTION MY_DATA.DAT -
_$ /SELECTION=SELECT_ALL -
_$ /BEGINNING=11:00 /ENDING=12:00 -
_$ /NOCLUSTER
%EPC-I-SCHEDULE, Entry MY_COLLECTION scheduled for 14-APR-1997 11:00
```

You can use the DURATION qualifier in place of the ENDING qualifier. You must specify the duration as a relative OpenVMS time.

For example:

```
$ COLLECT SCHEDULE COLLECTION MY_COLLECTION MY_DATA.DAT -
_$ /SELECTION=SELECT_ALL -
_$ /BEGINNING=11:00 /DURATION="+1::" -
_$ /NOCLUSTER
%EPC-I-SCHEDULE, Entry MY_COLLECTION scheduled for 14-APR-1997 11:00
```

3.2. How to Create a Report Based on Collected Data

You can use Oracle Trace software to produce reports that are based on the data collected from one or more Oracle Trace collections.

There are two steps to producing Oracle Trace reports:

1. Format and merge the data files.
2. Generate the report.

3.2.1. Formatting and Merging Data Files

Before Oracle Trace software can generate a report on the data collected for your collections, you must use the FORMAT command to format the data in the data files into an Oracle Rdb ™ database.[1]

The following example formats the data in the file MY_DATA.DAT and stores the formatted data in an Oracle Rdb database named FORMATTED_DATA.RDB:

```
$ COLLECT FORMAT MY_DATA.DAT FORMATTED_DATA.RDB
```

[1] A VAX RMS formatted file is also available for users who plan to create their own reports based on the data. For more information, see the Oracle Trace User's Guide.

You can also use the FORMAT command to combine the data files from two or more collections into one formatted database. However, these collections must have been scheduled by using the same facility selection.

There are two ways to combine data files: you can format several data files at once into the same Oracle Rdb database, or you can add a data file to an existing formatted database.

The following example combines the data collected from several collections into a new formatted database named WEEK.RDB.

```
$

COLLECT FORMAT MONDAY.DAT,TUESDAY.DAT,WEDNESDAY.DAT  WEEK.RDB
```

The following example adds the contents of the data file THURSDAY.DAT to the existing formatted database WEEK.RDB:

```
$ COLLECT FORMAT /MERGE THURSDAY.DAT WEEK.RDB
```

3.2.2. Generating a Report

Oracle Trace software can generate tabular reports based on the data in an Oracle Rdb formatted database.

Table 3.4 lists the three different types of reports that you can produce.

Table 3.4. Oracle Trace Reports

Type	Description
DETAILED	Actual values of the items collected for each event
FREQUENCY	Event occurrence summary based on a selected time interval
SUMMARY (default)	Summary statistics about the collected data

In the simplest case, the REPORT command creates a summary report on all the data in the formatted database and displays the report on your current SYS$OUTPUT device (usually your terminal).

For example:

```
$ COLLECT REPORT FORMATTED_DATA.RDB
```

You can further refine the report with the FACILITY qualifier.

The following example generates a summary report on only DECforms data and writes the report to the file MY_REPORT.TXT:

```
$ COLLECT REPORT FORMATTED_DATA.RDB /FACILITY=FORMS -
_$ /OUTPUT=MY_REPORT.TXT
```

You use the STATISTICS qualifier to specify the statistics for the Oracle Trace software to use in the summary report. This feature allows you to create customized reports that present the information in the manner most suitable to your needs.

The valid statistics are as follows:

- ALL (default)
- COUNT
- MAXIMUM
- MEAN
- MINIMUM
- NONE
- STANDARD_DEVIATION
- 95_PERCENTILE

The following example generates a summary report based on the data collected for DECforms software by using the statistics MAXIMUM, MINIMUM, and MEAN:

```
$ COLLECT REPORT FORMATTED_DATA.RDB /FACILITY=FORMS -
_$ /TYPE=SUMMARY /STATISTICS=(MAXIMUM,MINIMUM,MEAN) -
_$ /OUTPUT=MY_SUMMARY.TXT
```

By default, Oracle Trace software reports on the data collected for all the events contained in the formatted database. You can use the EVENTS qualifier to restrict the report to specific events.

The following example generates a report based on the data collected for a subset of the events in DECforms software:

```
$ COLLECT REPORT FORMATTED_DATA /OUTPUT=MY_REPORT.TXT -
_$ /FACILITY=FORMS -
_$ /EVENTS=(EVENT_1,EVENT_2,EVENT_3)
```

The BEFORE, INTERVAL, RESTRICTIONS, OPTIONS, and SINCE qualifiers provide additional customization features. For a full description of Oracle Trace reporting, see the Oracle Trace User's Guide.

3.3. Sample Oracle Trace Report for DECforms Software

Example 3.1 shows a frequency report produced by Oracle Trace for a DECforms application.

Example 3.1. Sample Frequency Report Using Oracle Trace Software

```
14-APR-1997 08:33              Frequency Report      Page 1
Selection: FORMS_EVENTS                              Oracle Trace V2.2

Event:  ACCEPT_PHASE    In Facility:  FORMS          Version:  V2.2

Time Period          Occurrences

14-APR-1997 08:13:00           5
14-APR-1997 08:14:00           7
14-APR-1997 08:15:00           2
14-APR-1997 08:16:00           1
14-APR-1997 08:18:00           2
14-APR-1997 08:19:00           4
```

```
%EPC-I-NOEND, 3 Start Event Records had no matching End

14-APR-1997 08:33              Frequency Report      Page 2
Selection: FORMS_EVENTS                              Oracle Trace V2.2

Event:  CALL_RESPONSE   In Facility:  FORMS          Version:  V2.2

Time Period          Occurrences

14-APR-1997 08:13:00           4
14-APR-1997 08:14:00          10
14-APR-1997 08:15:00           7
14-APR-1997 08:17:00           9
14-APR-1997 08:18:00          55
14-APR-1997 08:19:00           2

14-APR-1997 08:33              Frequency Report      Page 3
Selection: FORMS_EVENTS                              Oracle Trace V2.2

Event:  COLLECTION      In Facility:  FORMS          Version:  V2.2

Time Period          Occurrences

14-APR-1997 08:13:00           3
14-APR-1997 08:14:00           5
14-APR-1997 08:15:00           2
14-APR-1997 08:16:00           1
14-APR-1997 08:17:00           2
14-APR-1997 08:18:00          11
14-APR-1997 08:19:00           2

14-APR-1997 08:33              Frequency Report      Page 4
Selection: FORMS_EVENTS                              Oracle Trace V2.2

Event:  DISTRIBUTION    In Facility:  FORMS          Version:  V2.2

Time Period          Occurrences

14-APR-1997 08:13:00           4
14-APR-1997 08:14:00           8
14-APR-1997 08:15:00           1
14-APR-1997 08:16:00           3
14-APR-1997 08:17:00           5
14-APR-1997 08:18:00          10
14-APR-1997 08:19:00           3

14-APR-1997 08:33              Frequency Report      Page 5
Selection: FORMS_EVENTS                              Oracle Trace V2.2

Event:  PANEL_DURATION  In Facility:  FORMS          Version:  V2.2

Time Period          Occurrences

14-APR-1997 08:13:00           5
14-APR-1997 08:14:00          10
14-APR-1997 08:15:00           2
14-APR-1997 08:16:00           1
14-APR-1997 08:17:00           6
```

```
14-APR-1997 08:18:00          35
14-APR-1997 08:19:00           4

%EPC-I-NOEND, 3 Start Event Records had no matching End
%EPC-I-NOSTART, 1 End Event Records had no matching Start

14-APR-1997 08:33                    Index             Page 6
Selection: FORMS_EVENTS                                Oracle Trace V2.2

Report Index

Facility Name                Event Name                        Page

FORMS                        ACCEPT_PHASE                      1
FORMS                        CALL_RESPONSE                     2
FORMS                        COLLECTION                        3
FORMS                        DISTRIBUTION                      4
FORMS                        PANEL_DURATION                    5
```

Chapter 4. How the Form Manager Processes Requests

The Form Manager processes each request that you call from an application program by:

1. Initializing the request
2. Distributing form data (SEND and TRANSCEIVE requests only)
3. Processing external responses
4. Accepting input
5. Processing request exit response
6. Collecting form data (RECEIVE and TRANSCEIVE requests only)
7. Terminating the request

The Form Manager returns control to the application program after terminating the request. This chapter explains each phase of the request process.

4.1. Initialize Request Phase

The Form Manager performs the initialize request phase for each request that you call from your application program. During this phase, the Form Manager validates the request call.

4.1.1. Initializing Requests (Except ENABLE)

To initialize any request other than the ENABLE request, the Form Manager performs the following steps:

Step 1: Validating the Argument List

To validate the argument list in a request call, the Form Manager determines that the correct number of arguments was passed in the argument list and that each argument is the correct type. The Form Manager also verifies that it can read information from or write information to the appropriate arguments in the argument list.

Note

Because some programming languages allow you to store variables in either read-only memory, read/write memory (on some operating systems), or on the process stack, you must ensure that where you store a variable does not interfere with the Form Manager's access to that variable. For information on assigning variables to particular areas of memory, see the documentation for your programming language.

The Form Manager terminates the request and returns an error message to your application program if one of the following is true:

- The argument list contains too few or too many arguments (OpenVMS API only).

- An argument's data type is incorrect (OpenVMS API only).
- The Form Manager does not have the access it needs to an argument in the argument list.
- An illegal option is specified in the request-options argument.

Step 2: Validating the Session-Identification Argument

Once the Form Manager has determined that the argument list is valid, it validates the session-identification argument. The Form Manager maintains a list of active sessions. To validate the session-identification argument, the Form Manager compares the value in that argument to the list of active sessions.

If the value in the session-identification argument matches an active session, the argument is valid. The Form Manager uses the form and display device that are associated with the value of the session-identification argument for subsequent operations during this request.

If the session-identification string does not match an active session, the Form Manager terminates the request and returns an error message to your application program.

Step 3: Validating Record Names

After validating the session-identification argument, the Form Manager validates any record name (either send or receive) argument in the request. (This validation is not done for the DISABLE or CANCEL requests because you cannot pass a record name argument in either request.)

When the Form Manager validates the record name, it compares the value passed in that argument to the names of records stored in the form. The record name can specify either a single record name, or a list of records.

The record name argument must contain a character string that is the same as a form record name or record list name. For example, if you pass the string RECORD_ONE in the record name argument, there must be a form record or form record list named RECORD_ONE. (The Form Manager ignores case for form record names, so RECORD_ONE is equivalent to record_one.)

When the Form Manager finds a form record or record list that has the same name as the string passed in the record name argument, it validates the length of the data that is to be passed.

The Form Manager determines the length of that data by looking at the record that you pass in the request call. The Form Manager compares the length specified by the record message argument to the length of the form record in the form specified by the record name. If the length in the record and the length of the form record match, the record length is valid.

If the record name specifies a list, validation is performed the same way that it is performed for a single record for each record in the list. For example, if RECORD_ONE specifies a record list that contained 25 records, validation is performed 25 times, once for each record.

The Form Manager terminates request processing and returns an error message to your application program if one of the following occurs:

- The record-message-name argument does not match the name of any record defined in the form or record list.
- The length field of the record argument does not match the length of the form record that is named in the record-message-name argument.

Step 4: Validating Control-Text Arguments

Receive control text provides information on the status of a request to the application. The Form Manager passes receive control text to the application and the application passes send control text to the form. A control-text argument can be either receive control text or send control text, specified as a receive-control-text or send-control-text argument in all request types except CANCEL.

The Form Manager tests the send-control-text-count for read access and the receive control text for write access.

The Form Manager validates any control-text arguments during the validation of the argument list. Send control text passes the names of control text responses to the Form Manager. If you pass a control-text argument, either send or receive, you must also pass a control-text-count argument.

To validate a send-control-text argument, the Form Manager verifies that the value in the send-control-text-count argument is less than or equal to the number of send control text items specified in the send control text. (The send control text is a set of five-character items; each item is a five-character string.) The Form Manager does this by:

1. Reading the value of the send-control-text-count argument.

2. Multiplying this value by five.

3. Comparing the resulting value to the total number of characters in the send control text.

If the value in the send-control-text-count argument is greater than zero and less than or equal to the number of control text items in the send control text, the argument is valid. For example, if the count is 2, and there are 14 characters in the control text, the argument is valid (2 x 5 = 10 and 10 < 14). If the count is 3 and there are 14 characters in the control text, the argument is not valid.

If the send control text is invalid, the Form Manager terminates request processing and returns an error message to the application program.

The Form Manager ignores those control text items not present in the form.

Step 5: Validating Parent-Request-Ids

To validate a parent-request-id, the Form Manager first verifies that the parent-request-id matches the identification of an existing request.

The Form Manager terminates request processing and returns an error message to your application if one of the following occurs:

- The parent-request-id does not match the identification of an existing request.

- The parent-request-id is not awaiting completion of a procedural escape routine.

The Form Manager then verifies that the parent request awaits completion of a procedural escape routine.

Step 6: Validating Request Options

To validate the request options specified by a request call, the Form Manager matches the specified request options to the request. If the request specifies an option that is invalid for that request, validation

of the option fails and the Form Manager returns an error message and returns to the application program. The Form Manager also tests read and write access of the request options.

For more information on request options, see Chapter 5.

Step 7: Validating Shadow Records

Shadow records provide a means of tracking information concerning record fields. There are two kinds of shadow records: send shadow records and receive shadow records.

A send shadow record is sent from the application program to the form. The send shadow record contains an indicator that specifies whether the record fields in the associated send record should change the last known values of associated form data items.

You pass the send shadow record to DECforms by specifying the optional send-shadow-record argument on the SEND and TRANSCEIVE request calls.

A receive shadow record, which is a record returned to the application program, contains information about a record and each record field in the associated receive record. The information specifies whether the record fields in the associated receive record have changed value from the values last known to the program. A receive shadow record contains one character for each record field in the receive record plus one additional character that specifies the modified status of the record.

You pass the receive shadow record to DECforms by specifying the optional receive-shadow-record parameter on the RECEIVE and TRANSCEIVE requests.

You cannot specify shadow records for a CANCEL or DISABLE request.

4.1.2. Initializing the ENABLE Request

To initialize the ENABLE request, the Form Manager performs the following steps:

Step 1: Validating the Argument List

To validate the argument list in an ENABLE request call, the Form Manager determines that the correct number of arguments was passed in the argument list and that each argument is the correct data type. The Form Manager also verifies that it can read information from or write information to the appropriate arguments in the argument list.

Note

Because some programming languages allow you to store variables in either read-only memory, read/write memory (on some operating systems), or on the process stack, you must ensure that where you store a variable does not interfere with the Form Manager's access to that variable. For information on assigning variables to particular areas of memory, see the documentation for your programming language.

The Form Manager terminates the request and returns an error message to your application program if one of the following is true:

- The argument list contains too few or too many arguments.
- The Form Manager does not have the access it needs to an argument in the argument list.

Step 2: Creating the Session

To create a session, the Form Manager creates a unique, 16-character session-identification string. The session-identification string relates the display device that was attached and the form that was loaded into memory to each other. If the session-identification parameter is not 16 characters long, the Form Manager returns an error message and terminates request processing. (In the portable API, the 16-character string Forms_Session_Id is defined in the header file.)

Step 3: Loading the Form

Once the Form Manager has validated the argument list in an ENABLE request call, it locates the form and loads it. The Form Manager loads the form based on the parameters specified in the ENABLE request.

To load the form, the Form Manager performs the following tasks:

- If you specify FORMS$AR_FORM_TABLE and the form name parameter in the OpenVMS API, or if you specify form object in the request option of the portable API, the Form Manager checks to see if the form specified by the form name parameter is linked in with the application. If the form is linked in with the application, the Form Manager loads that form.

  If the form is not linked in with the application, and you specify both the form name and form file parameters, the Form Manager attempts to activate the file specified in the file name parameter and then looks for the form name specified in the form name parameter. If the Form Manager finds the form, it loads it.

- If you specified the file name and the form is not loaded in either the OpenVMS API or the portable API, the Form Manager attempts to load the file assuming that it is a valid form file. If a full file specification is not present, the current directory is the default. The default file type is .form.

If the Form Manager cannot find the form specified in the request call, it terminates request processing and returns an error to your application program.

The Form Manager refrains from loading a form file from disk if the file has been loaded by another session for this process. Rather than reloading the form, the Form Manager uses the current memory-resident version of the form for the session to reduce the overall memory requirements for the process.

The form file specifications passed to the ENABLE request must meet the following three criteria for a form to be shared between sessions within the same process:

- The full file specification of the form file must be identical to that of a loaded form.
- The file modification dates of the form files must match.
- The file sizes of the form files must match.

If no loaded memory-resident form meets all three conditions, the Form Manager reads and loads the form file from disk. The Form Manager deletes a memory-resident form file when no sessions reference it. This form load strategy is especially useful in the ACMS environment and in any environment where multiple instances of the application are used.

Step 4: Selecting the Layout

After the Form Manager loads a form, it selects a layout. To be selected, a layout must meet display device, device type, display size, and natural language requirements.

If the device name indicates a VT-class terminal, then the Form Manager attempts to select a matching VT or character-cell layout. A typical character-cell device name value is "SYS$INPUT".

If the device name indicates a disk file, then the Form Manager attempts to select a matching PRINTER layout. Valid PRINTER device name values are any legal file specification. An example would be "FORMS_OUTPUT.DOC".

To determine which layout in a form meets the display device requirement, the Form Manager first looks at the display device argument in the ENABLE request. In the display device argument, you pass the name of a display device. The Form Manager uses the information in this argument to retrieve information about the attributes of the display device; for example, whether the device is a VT300-series terminal, and if so, if it supports color.

In some cases, the operating system may not provide a way for DECforms to inquire about attributes of the display device. DECforms provides a way to specify default terminal settings.

To specify default terminal settings when you use the portable API, use the forms_c_opt_default_term and the forms_c_opt_default_color options. The fields for these options include:

default_color_type
default_term_type

Literals are defined in the API library files to simplify the assignment of these fields. The literals are in the format forms_c_dev_[xxx] and forms_c_color_[xxx]. The Form Manager uses the values you set as default values when it cannot determine the terminal type (or color support) from the operating system.

The layout must specify a display device type that matches the attributes that the Form Manager retrieves. If no layout specifies a display device type that matches the attributes of the physical display device, the Form Manager looks for the layout that most closely corresponds to the physical display device.

To determine whether a layout meets the display size requirement, the Form Manager compares the display size specified in the layout with the size of the physical display device. To meet the display size requirement, a layout must specify a display size that is valid for the physical display device. For information on valid display sizes for a particular display device, see the section on device declarations in the *VSI DECforms IFDL Reference Manual*.

To determine whether a layout meets the natural language requirement, the Form Manager checks if the FORMS$K_LANGUAGE item code or language_name option is specified. If you do not specify this item code or option, the Form Manager translates the FORMS$LANGUAGE logical name.

If the translation is successful, the layout is selected based on the value of this translation.

Alternatively, an application program can specify a language clause through the request options of an enable request. Specify item code FORMS$K_LANGUAGE in the OpenVMS API, and forms_c_opt_language in the portable API.

You cannot define FORMS$LANGUAGE to be a search list.

To meet the natural language requirement, a layout must specify the same natural language clause as is specified in the FORMS$K_LANGUAGE item code, the FORMS$LANGUAGE logical name, or FORMS_LANGUAGE variable. The character string specified in the layout must match either the character string to which FORMS$LANGUAGE or FORMS_LANGUAGE is defined or the FORMS $K_LANGUAGE item code; for example, if you defined FORMS$LANGUAGE to be “english”, the Form Manager would not select a layout that specifies “eng”.

When comparing the character string specified in a layout to the character string specified in FORMS $LANGUAGE or FORMS$K_LANGUAGE, the Form Manager ignores case.

Note

The Form Manager uses FORMS$LANGUAGE and FORMS$K_LANGUAGE to select a layout only. The logical name and item code do not cause the Form Manager to translate information in a layout to a new natural language clause; if literals in the layout are specified in English, they appear in English even if FORMS$LANGUAGE or FORMS$K_LANGUAGE and the layout specify German as the natural language clause.

You must provide form literals in the language clause in which you wish them to appear, regardless of whether you specify FORMS$LANGUAGE and FORMS$K_LANGUAGE.

The Form Manager ignores natural language when selecting a layout if it does not find one of the following:

- A specified FORMS$K_LANGUAGE item code
- A translation for the logical name FORMS$LANGUAGE

The Form Manager searches for a layout by looking at all the layouts specified in the form. If no layouts match the language clause first, then the Form Manager scans the layouts that do not specify a language clause.

VT Device Layout Selection

The Form Manager selects the layout that specifies the closest compatible device to the device specified in the enable call.

The list of supported VT devices follows in order of compatibility:

- VT100-No AVO
- VT100
- VT200
- VT200 Color
- VT300
- VT300 Color
- VT400

For example, if the enable device is a VT300 terminal and the form contains layouts for both the VT100 and VT400 terminals, the Form Manager selects the VT100 terminal because it is more closely compatible to the VT300. The Form Manager does not select the VT400 terminal because the VT400 capabilities are not compatible with terminals below the enable device, in this case the VT300 terminal.

To have only one layout to service all terminals above and including a VT100, specify a VT100 as the enable device. This layout is not compatible with the VT100-No AVO terminal.

The Form Manager evaluates VT layouts in the following order:

1. Vertical length—closest to without exceeding the vertical height of the enable device
2. Width—closest to without exceeding the width of the enable device

If more than one layout specifies the same *device type*, the Form Manager uses the vertical height (page size) of the enable device to resolve the conflict and selects the layout that specifies a vertical height closest to without exceeding that of the enable device.

If more than one layout specifies the same *vertical height*, the Form Manager uses the width of the layout to resolve the conflict.

If more than one layout equally satisfies all these criteria, the Form Manager selects the first such layout as defined in the IFDL.

PRINTER Layout Selection

When the Form Manager selects a PRINTER layout, it scans all %PRINTER layouts with a matching language clause for a SELECTION_LABEL match. If it finds no match, the Form Manager scans the set of %PRINTER layouts that do not specify a language clause for a SELECTION_LABEL match. If no match is found, the Form Manager chooses the first %PRINTER layout without a language clause in the form.

Step 5: Attaching the Display Device

Once the Form Manager has selected a layout, it attempts to attach the display device specified in the display device specification argument to the ENABLE request call. If the Form Manager cannot attach the specified display device, it terminates request processing and returns an error to your application program.

Step 6: Initializing Form Data Items

During the initialization of an ENABLE request, the Form Manager assigns initial values to all form data items, including **built-in form data items**. Built-in form data items are form data items to which only the Form Manager has write access.

During the initialization of an ENABLE request, the Form Manager stores the following values in the FORMNAME, SESSION, and TERMINAL built-in form data items (if they are declared in the form) as seen in Table 4.1.

Table 4.1. FORMNAME, SESSION, and TERMINAL Values

Form Data Item	Contents
FORMNAME	The character string name of the form in the IFDL file.
SESSION	The session identification string, which the Form Manager creates.
TERMINAL	The character string name of the display device that you specified in an ENABLE request or the translation if that string is a logical name.

If you declare built-in form data items in your IFDL source file, you can read values from them. You must declare built-in form data items to be CHARACTER or CHARACTER VARYING data types; you can specify any length for the data items and they can be character null-terminated.

If the value to be stored in any built-in form data items exceeds the length you declared for the form data item, the Form Manager truncates the value from the right.

If the value is shorter than the length you declared for the form data item and you declared the form data item to be the CHARACTER data type, the Form Manager pads the value on the right with spaces.

If you declared the built-in form data item to be CHARACTER VARYING, the length of the form data item becomes the length of the value, for as long as that value is stored in the item.

4.2. Data Distribution Phase

For SEND and TRANSCEIVE requests, the second phase of request processing is the data distribution phase. During this phase, the Form Manager copies data from application program record fields to form data items. This phase is not performed for any other request.

4.2.1. Determining Where Values Are Stored

The data passed in a SEND request or the send part of a TRANSCEIVE request comes from fields in an application program record. Therefore:

- Any application program record containing data that you want to pass to the form must be logically equivalent to a form record.
- The application program record must be the same length as a form record, and it must have the same number of fields as a form record.
- The fields in the application program record must have the same data type, length, and dimension as the corresponding fields in the form record.

Note

All program record data is expected by the Form Manager to be byte aligned.

The Form Manager determines which application program record and form record are logically equivalent by looking at the record name argument passed in a SEND or TRANSCEIVE request call. The order of the descriptors specified in the request call must match the order of the records in the record name. The character string in this argument must name the form record or records (specified in a record list) that are logically equivalent to the application program record storing the values that you are passing in the request.

The correspondence between the application program record and the form record allows the form record to act as a table of contents to the application program record. The form record does not store the values passed in a request. The values are stored in form data items. Therefore, fields in the form record must, in turn, correspond to form data items.

4.2.2. How the Data Is Distributed

When the Form Manager distributes form data, it moves the values from the application program record fields into form data items. If these form data items are displayed in panel fields, or used as panel object labels, or referenced in WHEN or FIRST clauses, the Form Manager also updates the values on the panel. (These form data items must have the same qualified names as their associated panel fields.)

If the form data items are not referenced in currently displayed panel fields, the Form Manager does not change the display.

More than one form record can be associated with each form.

Each form record consists of named record fields. A direct association is made between a record field and a form data item with the same qualified name. You associate the name of a record field with a form data item.

You can also define a panel field with the same name as the form data item to enable the operator to view the content of the record field. When no such panel field declaration exists, the application program sends and retrieves information that the operator cannot view.

You can override the default name association between record field and form data item by using the TRANSFER clause. This allows associations between form data and record fields other than by name. For more information on the TRANSFER clause, see the *VSI DECforms IFDL Reference Manual*.

If no association is established between a record field and a form data item, either by the default name association, or by use of the TRANSFER clause, the Form Manager performs a default action. For a send record, the Form Manager ignores the contents of a record field with no association.

Note

The fields in the application program record must have the same data type, length, and dimension as the corresponding fields in the form record. When data is distributed, the Form Manager distributes record field data to a form data item by converting data. During this conversion, if the data does not match, the Form Manager returns a data conversion error.

4.2.3. Using the DATA TRANSFER Clause

You use the DATA TRANSFER clause to specify that the name of a record field in a form record should be treated as if it had a different name (which must be the same as a form data item). The DATA TRANSFER clause acts as a bridge between convenient names for record fields and convenient names for form data items.

The SOURCE clause declares that the value of the record field is to be set from a form data item when the application program requests the given form record. The form data item can then broadcast its value to several record fields in the same form record.

The DESTINATION clause can specify that the record field value is to be copied to one or more form data items when the record is sent from the application program to the form, broadcasting its value to several form data items.

A record can have SOURCE and DESTINATION clauses pointing to different form data items to achieve special effects.

You can also use the DATA TRANSFER clause to move data and record field arrays. For more information on the DATA TRANSFER clause, see the *VSI DECforms IFDL Reference Manual*.

4.2.4. Shadow Records

In addition to the data contained within a form record, you can optionally have applications pass additional information that is associated with and describes special attributes of the record fields of the form record. The information is passed in a shadow record whose structure is directly related to the form record.

The special information is of two varieties, depending on the direction of the data transfer. Information about the modified status of the record fields can be returned to the application in a receive record, and the application can supply information as to whether last known values for modified field information should be updated.

Modified Fields

The Form Manager tracks the modified status of form data and returns this data to the application program if the program requests it in a shadow record. You might find this data useful in deciding whether certain actions must take place based on the data returned from the form.

Because keeping track of the modified status of form data will probably degrade performance, and because you may not need to know the modified status of all form data, you can track form data items individually, or track all the form data items in a data group, or all form data items in a form. The Form Manager saves the last known values of tracked form data items only.

Tracked data is associated with form data, which is directly connected to the record fields that are sent from and received into the program. The meaning of modified data depends upon the last known value of a form data item. The program expects a certain value for a form data item: the program knows what it has sent out to the form or what it received back from the form in a previous request. If the value being returned to the program in a receive record differs from this last known value, the value is marked as modified.

Modified data is maintained between requests. The program can send out a record field in a send request and ask for that record field back in a receive request later (perhaps several requests later).

If the value of the form data item that is associated with that record field returned has changed because of form activity (operator input, form procedural statements), the returned data states that the record field has changed.

If the value changed more than once, but the record retains its original value, the program receives data that the record field has not changed, because the value is associated with the last known value of the associated form data item.

Send Shadow Record

A send shadow record is a record that is sent from the application program to the form. The send shadow record contains an indicator specifying whether the record fields in the associated send record should change the last known values of form data items that were changed by the send record and are being tracked.

The send shadow record is passed by specifying the optional send-shadow-record argument on the SEND and TRANSCEIVE request calls. In the OpenVMS API, the shadow record must be passed by descriptor. In the portable API, use the Forms_Record_Data structure to set up the send shadow record.

There is no send-shadow-record-length argument in the OpenVMS API. The length is specified in the shadow descriptor. In the portable API, use the length field in the structure Forms_Record_Data.

The send shadow record consists of a single character. The length of the send shadow record must be 1. If the character is N (either uppercase or lowercase), the last known values of tracked form data items are not set when the form data items receive data from their associated form record fields during data distribution. If the first character is anything but N, or if a send shadow record is not specified on a SEND or TRANSCEIVE call, the last known values of form data items are set.

If a form data item is not specified as tracked, there are no associated last known values to be updated.

4.2.5. Data Conversion

If the data type of a program record field does not match the data type of its corresponding form data item, the Form Manager must convert the data in the program record field to the data type of the form data item. The Form Manager must also perform data conversion when a form data item is displayed in a panel field, and the form data item and panel field have the same qualified name during data distribution.

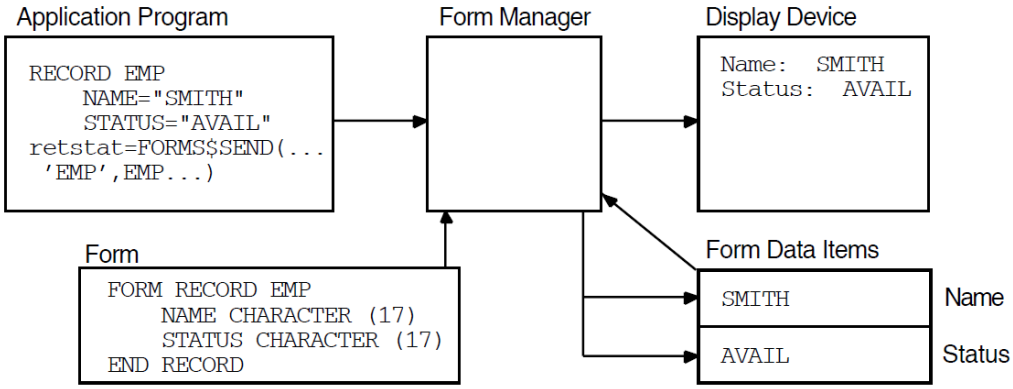
The Form Manager terminates request processing and returns an error to your application program if one of the following is true:

- The Form Manager cannot convert the data in a program record field to the data type of the form data item in which it attempts to store that data.
- The Form Manager cannot convert the form data item to a data type that can be displayed in a panel field.
- The data does not fit into the destination. In this case, a nonfatal conversion error is returned after the request has completed.

To distinguish the different conversion errors, set event logging or tracing and examine the output.

Figure 4.1 shows an exchange of information that might occur during a SEND request.

Figure 4.1. Function of the Form Manager During Data Distribution



The following steps describe the process outlined in Figure 4.1:

1. The SEND request tells the Form Manager to move the record data defined in the EMP program record to the EMP form record. The Form Manager looks in the form for a form record named EMP, and sees that EMP is composed of the form record fields NAME and STATUS.
2. The Form Manager moves the data to the NAME and STATUS form data items. If necessary, the Form Manager converts the data to the data type of the NAME and STATUS form data items.
3. The Form Manager takes the information in the form data items NAME and STATUS and displays it on the panel fields on the display device.

4.3. External Response Processing Phase

The third phase of request processing for SEND and TRANSCEIVE requests, and the second phase for other requests, is the external response processing phase. The Form Manager performs this phase for

each request you call from your application program. During this phase, the Form Manager processes responses and control text responses.

Section 4.3.3 explains the action that the Form Manager takes when it encounters each response step.

4.3.1. Performing Control Text Responses

One of the arguments that you can pass in a request call is send control text. Send control text can be composed of up to five **send control text items**. Each send control text item names one control text response stored in the form.

Each send control text item has a fixed five-character length. If a send control text item name is less than five characters, you must pad it with spaces. The number of valid send control text items is passed in a send control text item count argument or request option.

If a send control text item matches the name of a control text response stored in the form, the Form Manager performs that control text response. If there is no match in the form, that control text item has no effect.

You can use control text responses to cause the Form Manager to perform actions immediately before it performs the response to the request you called. By passing a send control text item, you can directly perform a particular response. For example, you could define a send control text response that applies a highlight to a set of fields that are about to be activated for input from the operator.

4.3.2. Performing Responses to Requests

Once the Form Manager completes any control text responses that you specify, it performs the response to the request. The Form Manager performs either a default response or a response you define.

Table 4.2 explains the default responses that the Form Manager performs in response to requests.

Table 4.2. Default Responses

Request	Default Form Manager Response
ENABLE	None.
DISABLE	None.
SEND	None.
RECEIVE	Perform the ACTIVATE CORRESPONDING RECEIVE ALL response step.
TRANSCEIVE	If a RECEIVE RESPONSE is defined for the record name in which the Form Manager stores data from the form, that response is performed as the default TRANSCEIVE response. If a RECEIVE RESPONSE is not defined for that record name, the Form Manager performs the ACTIVATE CORRESPONDING RECEIVE ALL response step.

If you want the Form Manager to perform actions other than those specified by the default response to an external request, you can declare an external response in your IFDL source file. Any response you declare overrides the default response.

For more information on defining responses, see the *VSI DECforms IFDL Reference Manual*.

4.3.3. Response Steps

Each response that you define consists of a set of one or more response steps. Each response step that you include in a response causes the Form Manager to perform a particular action. This section explains the processing that each response step causes the Form Manager to perform.

ACTIVATE Response Step

The ACTIVATE response step causes the Form Manager to add one or more activation items to the activation list for subsequent input processing. The ACTIVATE response step can specify the activation item as any of the following:

A field
A field array (or array element)
An icon
An icon array
A group
A group array (or array element)
A panel
All help panels for a field
A specific panel as a wait item
A wait without a panel
All panel fields

For more information about clauses you can specify with this response step to control what the Form Manager adds to the activation list, see the *VSI DECforms IFDL Reference Manual*.

If the Form Manager encounters the ACTIVATE response step during external response processing, it adds the necessary activation items to the bottom, or end, of the activation list. If the Form Manager encounters the ACTIVATE response step during the accept phase of processing, it inserts the necessary activation items in the activation list immediately following the current activation item.

For example, an activation list that consists of activation items for three fields might appear as follows:

Activation Item Number	**Field Name**
1	FIELD_A
2	FIELD_B
3	FIELD_C

If the Form Manager encountered an activate response step for a field named FIELD_D during the processing of the activation item for FIELD_B, the activation list would appear as follows:

Activation Item Number	**Field Name**
1	FIELD_A
2	FIELD_B
3	FIELD_D
4	FIELD_C

The Form Manager does not add two activation items for the same panel field, panel field array element, or wait to the activation list; each panel field or panel can correspond to only one activation item. (This also holds true for icons.) However, the activation list can contain any number of unique activation items.

When the Form Manager activates whole panels or groups within panels, then panel fields and icons are added to the activation list in the order in which they are defined in the panel definition (not the order in which the fields and icons appear on the display device).

If all the panel fields are stored in a single panel, the Form Manager adds activation items in the order in which the panel fields are stored in the definition of that panel. If the panel fields appear on more than one panel, the Form Manager adds activation items for all the panel fields stored in the first panel in the form.

Then it adds activation items for all the panel fields stored in the second panel in the form, and so on, until all the necessary activation items have been added.

For example, suppose that the following response step appears in your IFDL source file:

```
ACTIVATE FIELD CLIENT ON PANEL_ONE
         GROUP EMPLOYEE_INFO ON PANEL_TWO
```

Now suppose that CLIENT is a panel field on panel ONE and that EMPLOYEE_INFO is a panel group on panel TWO. The EMPLOYEE_INFO group contains panel fields named STATUS, ID_NUMBER, and PHONE (in that order).

The following table shows how the Form Manager would add activation items to the activation list in response to the ACTIVATE response step shown in the preceding example:

Activation Item Number	Panel Field Name	Panel Name
1	CLIENT	CLIENT_PANEL
2	EMPLOYEE_INFO.STATUS	STATUS_PANEL
3	EMPLOYEE_INFO.ID_NUMBER	NUMBER_PANEL
4	EMPLOYEE_INFO.PHONE	PHONE_PANEL

CALL Response Step

The CALL response step causes the Form Manager to transfer control to an escape routine. You can use the following mechanisms to specify arguments in the response step to pass to the escape routine:

- BY REFERENCE
- BY DESCRIPTOR
- BY VALUE
- BY DEFAULT

The CALL response step uses copy-in, copy-back semantics for passing arguments. When you pass a form data item BY REFERENCE, you do not pass the address of the form data item itself: a copy of the form data item is created and that address is passed to the escape routine. This temporary data item is copied back to the form data item when the escape routine returns.

If the escape routine modifies an argument that corresponds to a form data item, the Form Manager modifies the form data item upon return to the Form Manager from the escape routine. When control returns from the escape routine, the Form Manager continues processing the request.

For more information on the call response step and form data items, see the *VSI DECforms IFDL Reference Manual*. For information on writing and calling escape routines, see Chapter 2.

DEACTIVATE Response Step

The DEACTIVATE response step causes the Form Manager to remove one or more activation items from the activation list. The DEACTIVATE response step can specify the activation item as any of the following:

A field
A field array (or array element)
An icon
An icon array
A group
A group array (or array element)
An entire panel
All fields on the activation list
A wait for a specific panel
A wait without a panel.

For more information about the optional clauses you can specify with this response step, see the *VSI DECforms IFDL Reference Manual*.

If the activation item being removed is not the current activation item, it is immediately removed from the activation list.

If the activation item that is removed is the current activation item, the Form Manager completes the processing of that activation item by processing any field, group, or panel exit responses. Only after the applicable exit responses have been performed does the Form Manager remove the activation item from the activation list.

If the Form Manager does not find a specified activation item on the activation list, it ignores this item, but puts a message in the trace file (if tracing is turned on) and continues processing the response step.

DISPLAY Response Step

The DISPLAY response step causes the Form Manager to display the specified panels on the display device.

For PRINTER devices this response step causes the Form Manager to display the DDIF representation of a panel to the output DDIF document.

For the "%PRINTER" device on Microsoft Windows, this response step causes the Form Manager to spool a representation of the panel to the Microsoft Windows' default printer.

If you specify a viewport in the response step, the Form Manager displays the panel in that viewport.

If a panel has already been displayed, but becomes obscured or covered up, the DISPLAY response step redisplays the panel.

If you do not specify a viewport in the response step, the Form Manager uses the viewport specified in the panel definition.

If you did not specify a viewport in either the response step or the panel definition, the Form Manager displays the panel in the default viewport for the selected layout.

On PRINTER layouts, the DISPLAY response step with the IMMEDIATE clause specifies that the currently open DDIF file be closed after the panels are downloaded to the DDIF file. The IMMEDIATE clause with the DISPLAY response step has no meaning in character-cell layouts.

ENTER HELP Response Step

The ENTER HELP response step initiates help processing. The current location on the main activation list is saved, and the help activation list is created.

After the help activation list is created, accept phase processing restarts on that list. The HELP ACTIVE elementary condition is set to true. The values of the CURRENTITEMHELPED and CURRENTPANELHELPED built-in form data items are set to the names of the current activation item (if a field or icon) and its panel.

EXIT HELP Response Step

The EXIT HELP response step sets the HELP ACTIVE elementary condition to false and switches from the help activation list to the main activation list. It does not change the current position on the main activation list. If IMMEDIATE is specified, validation is not performed on the current HELP field.

The EXIT HELP response step causes the Form Manager to terminate accept phase processing for the help activation list. The Form Manager continues processing of the current activation item on the help activation list. When the current activation item passes validation, the Form Manager begins the termination check stage of accept processing.

Once the termination check is complete, accept phase processing for the help activation list finishes. The Form Manager resumes processing on the main activation list.

The IF Response Step

The IF/THEN/ELSE response step allows optional response steps to be performed based on the value of the conditional expression specified in the IF clause. When the condition is true, the Form Manager performs the response steps specified in the THEN clause. When the condition is false, the Form Manager performs the response steps specified in the ELSE clause. When the condition is false and no ELSE clause exists, the response step has no effect.

For more information on conditional expressions, see the *VSI DECforms IFDL Reference Manual*.

INCLUDE Response Step

The INCLUDE response step names an internal response that is to be performed as part of the processing for the current response. The Form Manager invokes internal responses only when it encounters an INCLUDE response step.

Use internal responses to define response steps that are used in several places in the form. Defining internal responses to perform common processing functions can result in smaller forms and improved performance.

INVALID Response Step

When the Form Manager performs accept responses (entry, exit, function, and validation responses), the INVALID response step causes the validation for the current activation item to fail. This response step also negates any POSITION response step and causes the Form Manager to perform an implicit POSITION IMMEDIATE TO CURRENT response step.

If the Form Manager is not in the accept input phase of processing, it ignores the INVALID response step.

LET Response Step

The LET response step causes the Form Manager to assign a value to a form data item. The value assigned can be a literal, or the value of another form data item.

The operands that you specify must conform to the data type conversion rules specified in the *VSI DECforms IFDL Reference Manual*. If you specify an operand that does not conform to these rules, the Form Manager terminates the request and returns an error message to your application program.

MESSAGE Response Step

The MESSAGE response step causes the Form Manager to display a character string in the message panel. If the message panel is not displayed, the Form Manager performs an implicit DISPLAY PANEL response step before it displays the character string in the message panel.

The Form Manager supports a single message panel for each layout in a form. This message panel has no panel fields or literals defined. The panel's width and height are the width and height of its associated viewport. It is used by the Form Manager to display information from MESSAGE and MESSAGE HELP response steps and from operator entry errors.

The Form Manager creates a default message panel in a default message viewport at form load time if none is defined for the selected layout. For character-cell layouts, the default viewport for the message panel is the width of the layout. The height is one line.

PRINTER layouts do not support message panels, therefore no default message panel is created.

The Form Manager treats the message panel like a scrolled region. The message panel is divided into lines and columns based on the size of the message panel and the size of the font assigned for that message panel.

Each message character string is written to the bottom of the message panel in a left-to-right direction on the bottom line of the panel and in a right-to-left direction in Hebrew layouts.

On character-cell devices, if the message character string is longer than the width of the panel, the string is word-wrapped to the next line. The current line in the message panel is scrolled up one line. The remaining characters in the message string are then written to the message panel using word-wrapping.

A word is any group of contiguous characters that lies between two word boundary characters. The word boundary characters are:

- SPACE
- LINE_FEED
- VERTICAL_TAB
- FORM_FEED
- CARRIAGE_RETURN

These characters are displayed, not interpreted, as control characters in this environment. If no boundary characters are found in the string, the string is broken at the length boundary of the line. The remaining characters, including the boundary character, are displayed on the next line using the previously discussed rules.

Word-wrapping of messages is supported only in character-cell layouts.

New messages are displayed at the left margin of the bottom line. Messages are displayed at the right margin in Hebrew layouts. If the bottom line is not blank, the contents of the message panel are shifted up as previously described.

You can specify the contents of the message that you want to display in the following ways:

- Pass an application program record field called MESSAGEPANEL. The contents of the record field are displayed in the message panel on the display device as soon as the Form Manager receives the record field. The application program record field must be declared to be either the CHARACTER, CHARACTER VARYING, or CHARACTER NULL STRING data type.
- Include a string literal in the MESSAGE response step.
- Include the name of a form data item in the MESSAGE response step. This causes the Form Manager to display the value of the form data item.
- Specify the HELP keyword. The Form Manager displays a help message associated with the current activation item. This help message may be declared at the panel field, panel group, panel, or layout level of the form.
- Include one of the Form Manager Message Codes, shown in the Table 4.3.

Table 4.3. Form Manager Message Codes

%CANTMOVELEFT	%MIN_LEN_FAILS	%NO_MORE_HELP
%CANTMOVERIGHT	%NODOWNOCC	%NO_NEXT_ITEM
%DATA_CONVERSION	%NOLEFTOCC	%NO_PREV_ITEM
%ENTRY_REQ_FAILS	%NORIGHOCC	%NO_RIGHT_ITEM
%FIELD_FULL	%NOUPOCC	%NO_UP_ITEM
%FUNCTIONS_ONLY	%NO_DOWN_ITEM	%RANGE_FAILS
%HELP_INACTIVE	%NO_HELP_AVAIL	%REQUIRE_FAILS
%INVALID_DATE	%NO_LEFT_ITEM	%SEARCH_FAILS

OpenVMS Online help contains explanations of these codes.

To read these explanations, enter a command in the following format: on OpenVMS systems:

```
$ HELP FORMS ERRORS message-ident
```

Message-ident is the abbreviation of the message. For example:

```
$ HELP FORMS ERRORS NO_UP_ITEM
```

POSITION Response Step

The POSITION response step designates which item in the activation list the Form Manager will process after completing the current activation item. Neither the activation list nor the current activation item is modified by this response step; only the designation of the next activation item is modified.

If the Form Manager cannot find an activation item that meets the requirements of a POSITION response step, it does nothing. The Form Manager does not return an error message. For example, if you

specify POSITION TO FIRST_FIELD and FIRST_FIELD does not correspond to any activation item, the Form Manager does nothing. If tracing is enabled, the Form Manager puts a message in the trace file stating that it cannot find the specified activation item.

A *relative* POSITION response step is any POSITION response step that does not name:

A specific panel
A specific field
A specific icon
A specific group
FIRST ITEM
LAST ITEM
FIRST PANEL
LAST PANEL

If you are specifying a relative POSITION response step, and there is no active item, the Form Manager ignores the POSITION response step and puts a message in the trace file. This result commonly occurs if you specify relative POSITION response steps inside an external request response before the accept phase begins.

IMMEDIATE specifies that the Form Manager not complete the current item's validation response for the current activation item before moving to the specified activation item. However, the Form Manager does complete the processing of the response that contains the POSITION IMMEDIATE response step and any exit responses for the current activation item. You can use the IMMEDIATE clause with any other POSITION response step clause.

For more information about the optional clauses you can specify with this response step, see the *VSI DECforms IFDL Reference Manual*.

If you specify a POSITION response step followed by another POSITION or POSITION IMMEDIATE response step, the latter response step overrides the former. When you specify the POSITION IMMEDIATE response step, subsequent POSITION response steps are ignored: the POSITION response step can be overridden by subsequent POSITION and POSITION IMMEDIATE response steps, but the POSITION IMMEDIATE response step can be overridden only by subsequent POSITION IMMEDIATE response steps.

PRINT Response Step

The PRINT response step causes the Form Manager to print the display contents of the specified panels. If no panels are specified in the response step, all panels displayed by the current session are printed. Each PRINT response step causes a separate page to be printed. Panel fields and all form literals are sent as part of the panels. If a panel name is specified, the specified panel or panels are downloaded to a file for printing.

The PRINT response step with the IMMEDIATE clause specifies that the currently open output file be closed after the panels are downloaded to the print file. The file specification of the print file is specified by the FORMS$PRINT_FILE logical name or the FORMS$K_PRINTFILE item code in the OpenVMS API and forms_c_opt_print option in the portable API. (For more information, see Chapter 5 and Chapter 6.) This response step is valid only in character-cell layouts. To print panels in PRINTER layouts, use the DISPLAY clause.

In Microsoft Windows, the PRINT Response step causes the Form Manager to print the display contents of the specified panels to the current default printer, using the characteristics most recently selected during the Printer Setup or from the application's system menu. You can use the Printers function of the

Microsoft Windows Control Panel application to select a default printer and set up printer characteristics, such as page size and orientation.

For more information on configuring printers and using the Control Panel to change printer configurations, refer to the appropriate Microsoft Windows documentation.

The PRINT response step is independent of the application program; this response step allows the form to print a panel directly without interacting with the application program. For an example of printing the current panel, see the *VSI DECforms Guide to Developing an Application*.

REFRESH Response Step

The REFRESH response step specifies that a designated viewport or viewports on the display be repainted.

You can specify two options for the REFRESH response step. ALL designates that all viewports for the current layout be refreshed on the display, and Viewport-name designates that the specified viewport or viewports be refreshed on the display.

If you do not specify one of the preceding options, the Form Manager designates that the contents of the default viewport for this layout are to be refreshed.

REMOVE Response Step

The REMOVE response step causes the Form Manager to delete the specified viewports from the display. This removes the contents of panels displayed on the viewport.

You can specify any of the following optional clauses when you use the REMOVE response step:

- ALL specifies that all viewports for the current session be deleted. The Form Manager clears the display when REMOVE ALL is specified.
- Viewport-name specifies that the named viewport be removed from the display.
- HELP specifies that all viewports displayed during help processing be removed.

If you do not specify one of the preceding options, the Form Manager deletes the default viewport for this layout. If your layout is a framed layout, the Form Manager deletes the Frame viewport.

RESET Response Step

The RESET response step causes the Form Manager to set the values of the specified form data items to their initial values. The initial value of a form data item is the value specified in the FORM DATA declaration: spaces if the form data item is a character data type, or zeros if the form data item is a numeric data type.

You can specify that a particular form data item, a group of form data items, or all form data items be reset. If any of the form data items that the Form Manager modifies correspond to currently displayed panel fields, the Form Manager also updates the display.

RETURN Response Step

The RETURN response step causes the Form Manager to terminate accept phase processing. The Form Manager continues processing the current activation item as normal. If the current activation item fails validation, the Form Manager begins the operator input stage of accept processing again.

When the item is validated, the Form Manager begins the termination check stage of accept phase processing. Once the termination check stage finishes, the Form Manager completes processing of the accept phase.

If you specify the IMMEDIATE clause with the RETURN response step, the Form Manager terminates the accept phase without performing validation at the field, icon, group, or request levels. It does not perform any remaining entry responses for the current activation item. However, the Form Manager does perform exit responses at the field, icon, group, and panel levels.

You can specify a receive control text item with the RETURN response step that will be returned to your application program as part of the receive control message. Section 4.7 describes the format of receive control text messages.

If you specify a RETURN response step followed by another RETURN or RETURN IMMEDIATE response step, the latter response step is performed. However, when you specify the RETURN IMMEDIATE response step, all subsequent RETURN or RETURN IMMEDIATE response steps are ignored: the RETURN response step can be overridden by subsequent response steps, but the RETURN IMMEDIATE response step can never be overridden by subsequent response steps.

SIGNAL Response Step

The SIGNAL response step causes the Form Manager to send a signal to the display device. Depending on your display device and the type of signal specified, the SIGNAL response step causes the Form Manager to ring the terminal bell or put the screen into reverse video.

VALIDATE Response Step

The VALIDATE response step causes the Form Manager to validate the specified items on the activation list. If an item specified is not on the activation list, it is ignored.

If any named item is a grouping (array, group, panel), only those elements of the grouping on the activation list are validated. The validation response associated with the grouping is also executed. The validations performed are all declared validation clauses for the relevant items. VALIDATE response steps for those items are not executed.

The VALIDATE response step is ignored if one of the following response steps has been executed in the current context. These response steps turn off validation:

POSITION IMMEDIATE
RETURN IMMEDIATE
EXIT IMMEDIATE
INVALID

The Form Manager performs the validation as if it had validated the item during the accept phase.
If the Form Manager detects a validation failure or a POSITION IMMEDIATE at the end of a field validation or at the end of any validation response, it stops further validation. A response can determine that validation failed by determining that the elementary-condition IMMEDIATE is true.

The Form Manager ignores recursive validate steps. If the Form Manager encounters a validate step during the execution of another validate step, the Form Manager ignores the second validate step.

During validation of an activation item, the Form Manager temporarily sets certain data items to values they would hold if the Form Manager were conducting validation of the item. After the completion of the validation step, the Form Manager restores such data items to the values they held prior to execution of the validate step. The data items affected are as follows:

- CURRENTITEM, CURRENTPANEL, FIELDVALUE, FIELDIMAGE, LOCATORITEM, and LOCATORPANEL
- The CURRENT data items for arrays

The Form Manager performs validation for activation items depending on their types. For fields, the Form Manager executes the field property validations and then zero or more validation responses. For icons, the Form Manager executes only the validation response for the object.

The Form Manager also executes one or more validation responses for each validation step. The Form Manager always executes an icon or field validation response for icons or fields.

If the validate step specifies an entire grouping of items, rather than a selection of items from a grouping, the Form Manager also a executes validation response for the grouping.

For more information about the VALIDATE response step, see the *VSI DECforms IFDL Reference Manual*.

4.4. Accept Phase

The fourth phase of request processing for SEND and TRANSCEIVE requests, and the third phase for other requests, is the accept phase. During the accept phase, the Form Manager processes the activation list. The Form Manager performs this phase for each external request you call that requires operator input. An external request requires operator input when items are activated. If no items are placed on the activation list, no accept phase processing is done.

The activation list is composed of activation items, which are satisfied by operator input.

The four types of activation items and the operator entries that satisfy them are as follows:

- Field activation items, which correspond to panel fields, are satisfied by both data input and function input from the operator.
- Icon activation items, which correspond to icons, are satisfied by function input from an operator. An icon is an element, much like a panel field, that can contain graphics or text; however, the operator cannot enter data in an icon.
- Wait activation items, which may or may not correspond to panels that are to be displayed, are satisfied by function input from the operator.

Activation items can be added to the activation list before or during accept phase processing.

If the current activation item is a wait or icon, the Form Manager accepts only function entry from the operator. No data characters are accepted.

If the current activation item is a panel field, the Form Manager accepts input and functions from the operator:

- For slider fields, the input is in the form of locator actions or keyboard functions that modify the value of the slider field. A locator is a device, such as a mouse, that the operator can use to navigate the form.
- For picture fields, the data characters are validated against the input picture defined for the panel field.
- For text fields, the data is entered into the field with no picture character validation.

Text fields can contain multiple lines of data.

Each input picture is composed of one or more edit picture characters. These edit picture characters define the character range permitted for the particular data position in the field. Validation of input is performed differently for different layouts:

- Character-cell layouts

  Character-by-character validation is performed; the operator's input is validated against the edit picture character as each character is typed. Insertion literals need not be typed. If the character is not acceptable, an error is signaled and a message is displayed in the message panel.

  For more information about edit picture characters, see the *VSI DECforms IFDL Reference Manual*.

- PRINTER layouts

  Because PRINTER layouts are output only, the Form Manager never enters the accept phase when processing PRINTER layouts.

The Form Manager starts the accept phase by determining which activation item it should process first. Usually, the Form Manager processes the first item on the activation list. However, if you specify POSITION response steps that are performed during the external response processing phase, these response steps control which activation item the Form Manager processes first.

When the Form Manager encounters POSITION response steps during the external response phase, it begins the accept phase by processing the item specified in the last POSITION response step performed during the external response processing phase.

The POSITION response step does not modify the order of items on the activation list. The POSITION response step controls only which activation item is processed first.

The Form Manager continues the accept phase by processing each activation item on the activation list. The POSITION response steps that the Form Manager encounters during the accept phase control the order of activation item processing.

Each function that the operator invokes to satisfy an activation item should contain a POSITION response step. These response steps indicate to the Form Manager into which item the operator will enter input next. Certain conditions can alter the order in which the Form Manager processes the activation list. Section 4.4.16 discusses these conditions.

4.4.1. Form Data Assignment Stage

The first stage in activation item processing is the form data assignment stage. During this stage, the Form Manager assigns values to the following built-in form data items:

- CURRENTITEM
- CURRENTPANEL
- FIELDIMAGE
- FUNCTIONNAME
- LOCATORITEM
- LOCATORPANEL

If the Form Manager is processing a field or icon, it stores the name of the item for which input is needed in the CURRENTITEM and LOCATORITEM built-in form data items. It also sets the FIELDIMAGE built-in form data item for picture and text fields to the value that is displayed in the field.

If the Form Manager is processing a wait activation item, it stores the name of the panel to be displayed in the CURRENTITEM, LOCATORITEM, LOCATORPANEL, and CURRENTPANEL built-in form data items and sets the FIELDIMAGE and FUNCTIONNAME built-in form data items to spaces. If the Form Manager is processing a wait activation item that is not associated with a panel, it sets the CURRENTITEM, LOCATORITEM, LOCATORPANEL, and CURRENTPANEL built-in form data items to spaces.

4.4.2. Panel Entry Response Stage

If the current activation item is a panel, field or icon on a panel, the Form Manager looks for a panel entry response for that panel.

The Form Manager performs any panel entry response you declared for that panel if one of the following is true:

- The accept phase is just beginning.
- The current activation item is a panel other than the panel that is associated with the previous activation item.
- The current activation item is contained in a panel that is different from the panel that contains the previous activation item processed.

  For example, suppose the previous activation item is a panel field on panel ONE. If the current activation item is a panel field on panel TWO, the Form Manager performs the panel entry response stage for panel TWO.

If you do not declare a panel entry response, the Form Manager skips this stage of activation item processing.

4.4.3. Group Entry Response Stage

If the current activation item is a member of a group, the Form Manager performs any entry response you declared for that group when one of the following is true:

- The accept phase is just beginning.
- The item that is the current activation item belongs to a different group than the item that is the previous activation item.

  For example, suppose the previous activation item is a panel field in group A. The Form Manager performs the group entry response for group B when the current activation item is a panel field in group B.
- The previous activation item does not belong to this group.

If you do not declare a group entry response, or if the current activation item is not in a group, the Form Manager skips this stage of activation item processing.

Fields, and icons can be nested within up to two levels of groups. When processing group entry responses, the Form Manager executes entry responses for each group that is being entered relative to the

previous activation item. The order of execution of entry responses is from outermost group to innermost group.

4.4.4. Field Entry Response Stage

If the current activation item is a panel field for which you declared an entry response, the Form Manager performs that response immediately before soliciting input from the operator.

If you do not declare an entry response for the field or icon, the Form Manager skips this stage of activation item processing. If you are performing input on the same item as the previous item (as is the case if validation fails), the Form Manager also skips this stage of activation item processing.

4.4.5. Operator Input Stage

During the operator input stage, the Form Manager accepts operator input to satisfy the activation item.

If the Form Manager is processing a field activation item, the operator can enter data in the panel field and issue functions to satisfy the activation item. As the operator enters data, the Form Manager validates the data.

On character-cell devices, the Form Manager interprets picture strings and the editing clause for numeric data items as the operator enters data. If the user enters a character that does not match that specified by the picture string, the Form Manager outputs a message to the message panel indicating an error. The Form Manager then continues to accept input data.

After operator input, the FIELDIMAGE built-in form data item is modified to reflect the operator input.

When the operator enters a function to complete entry into the panel field or icon that is the field activation item, the Form Manager stores the name of the function in the FUNCTIONNAME built-in form data item.

The Form Manager does not change the values in the FIELDIMAGE built-in form data item when it is processing a wait or an icon.

Any error messages generated by data or functions that the operator enters are displayed in the message panel on the display device; these errors do not terminate request processing.

Before the Form Manager can exit from the operator input stage, the operator must enter a valid function other than one of the following intra-field editing functions:

- CURSOR LEFT
- CURSOR RIGHT
- CURSOR UP
- CURSOR DOWN
- DELETE CHARACTER
- INSERT OVERSTRIKE
- ERASE FIELD

For information about editing functions, see the *VSI DECforms IFDL Reference Manual*.

4.4.6. Data Conversion Stage

If the activation item being processed is a field activation item, the Form Manager attempts to convert the field's value displayed in the panel field to the data type of its corresponding form data item.

If the conversion is unsuccessful, the Form Manager displays a message on the message panel. The operator input stage begins again unless the IMMEDIATE clause has been specified with a RETURN or POSITION response step.

If the IMMEDIATE clause has been specified, the Form Manager does one of the following:

- Returns control to the application program, as specified by the RETURN IMMEDIATE response step
- Begins processing the activation item specified in the POSITION IMMEDIATE response step

If the current activation item is not a field activation item, the Form Manager skips this stage of activation item processing.

4.4.7. Function Response Stage

During the function response stage, the Form Manager performs the default response steps specified for the function the operator entered. You can specify actions other than this by defining a special function response. The *VSI DECforms IFDL Reference Manual* describes special function responses.

4.4.8. Field Validation Stage

During the field validation stage, the Form Manager performs any validation that you specified for the item in the item declaration of your IFDL source file. For example, you may have specified that the Form Manager checks that the picture field is of a specified minimum length, that the data and function entry meet the requirements of the panel field, and so on. If any validation clause fails, the Form Manager displays a message on the message panel and performs an implicit INVALID response step.

If the current activation item is not a field or icon activation item, the Form Manager skips this stage of activation item processing.

4.4.9. Field Validation Response Stage

During the field validation response stage, the Form Manager performs any validation response that you declared for the field. In a validation response, you can specify validation beyond that specified in the field declaration. If the Form Manager encounters an INVALID response step when it is processing a validation response, it does not perform any further validation responses. However, it does process any exit responses for the field.

When the Form Manager is processing a field or icon activation item, it performs the field validation response stage if one of the following is true:

- The operator has entered a function that contains a POSITION response step that indicates another activation item as the next activation item to be processed.
- Accept phase processing is being terminated.

If the Form Manager does not encounter an INVALID response step while processing this stage, it assumes that the item is valid.

If you did not declare a validation response, if the current activation item is not a field or icon activation item, or if the next activation item is the same as the current activation item, the Form Manager skips this stage of activation item processing.

4.4.10. Field Exit Response Stage

When the Form Manager is processing a field, or an icon activation item, it performs the field exit response stage if one of the following is true:

- The operator has entered a function that contains a POSITION response step that indicates another activation item as the next activation item to be processed.
- Accept phase processing is being terminated.

The Form Manager performs the exit response even if it is processing a POSITION IMMEDIATE or RETURN IMMEDIATE response step.

If the Form Manager is not processing a field or icon activation item, or if you did not declare an exit response for the item, the Form Manager skips this stage of activation item processing.

4.4.11. Group Validation Response Stage

During the group validation response stage, the Form Manager performs any validation response you declared for a group if one of the following is true:

- The next activation item is in a group other than the group that contains the current activation item.

  For example, suppose the current activation item is a panel field in group A. The Form Manager performs the group validation stage for group A when the next activation item to be processed is a panel field in group B.
- The next activation item is an entity that is not in a group.
- The Form Manager is terminating accept phase processing.

If the Form Manager encounters an INVALID response step when it is processing a group validation response, it does not perform any further validation responses. However, it does process any exit responses that you declared for the item.

If the Form Manager does not encounter an INVALID response step while processing this stage, it assumes that the group is valid.

If you did not declare a group validation response or if the current activation item is not in a group, the Form Manager skips this stage of activation item processing.

4.4.12. Group Exit Response Stage

If the current activation item is a field or icon that is a member of a group, the Form Manager performs any exit response you declared for that group if one of the following is true:

- The next activation item is an item in a group other than the group that contains the current activation item.

  For example, suppose the current field activation item is a panel field in group A. The Form Manager performs the group exit response for group A when the next field activation item to be processed is a panel field in group B.

- The next activation item is an item that is not in a group.
- The Form Manager is terminating accept phase processing.

The Form Manager performs the group exit response even if it is processing a POSITION IMMEDIATE or RETURN IMMEDIATE response step.

For nested groups, group validation and group exit responses execute in turn, inner group first, then outer group, for each level of nested group being exited.

If you did not declare a group exit response or if the Form Manager is not processing an item in a group, the Form Manager skips this stage of activation item processing.

4.4.13. Panel Validation Response Stage

During the panel validation response stage, the Form Manager performs any validation response you declared for a panel if both of the following are true:

- The current activation item is active.
- No INVALID, POSITION IMMEDIATE, EXIT IMMEDIATE, or RETURN IMMEDIATE was executed.

If any of the following are true, the Form Manager interprets the panel validation response of the panel of the current activation item:

- The next item is on another panel.
- The next item is not associated with a panel (it is a wait without a panel).
- An EXIT IMMEDIATE or a RETURN IMMEDIATE response step was executed.
- An EXIT or RETURN response step without IMMEDIATE has been executed previously and no INVALID step has been executed since.

If the Form Manager encounters an INVALID response step when it is processing a panel validation response, it does not perform any further validation responses. However, it does process any exit responses that you declared for the item.

If the Form Manager does not encounter an INVALID response step while processing this stage, it assumes that the panel is valid.

If you did not declare a panel validation response or if the current activation item is not in a panel, the Form Manager skips this stage of activation item processing.

After interpreting the response, the Form Manager goes to the next activation item.

4.4.14. Panel Exit Response Stage

During the panel exit response stage, the Form Manager performs the exit response you declared for the panel associated with the current activation item if one of the following is true:

- The next activation item is a panel other than the panel that is the current activation item.
- The next activation item is on a different panel than the panel that contains the item that is the current activation item.

For example, suppose the current activation item is a panel field on panel ONE. The Form Manager performs the panel exit response stage for panel ONE when the next activation item is a panel field on panel TWO.

- The Form Manager is terminating accept phase processing.

The Form Manager performs the panel exit response even if it is processing a POSITION IMMEDIATE or RETURN IMMEDIATE response step.

If you did not declare a panel exit response, the Form Manager skips this stage of activation item processing.

4.4.15. Termination Check Stage

The Form Manager performs the termination check stage if both of the following are true:

- A RETURN response step was executed in a previous response.
- There are items on the activation list.

During the termination check stage, the Form Manager validates each item on the activation list. This process includes performing field validation clauses, and performing validation responses for fields, icons, groups, and panels. Validation clauses and validation responses can be performed multiple times for each item on the activation list, because each item can be validated while it is active and when the accept phase is terminated.

During this stage, the Form Manager also performs any validation response that you defined in the REQUEST VALIDATION external response. If the activation list is empty, no validation checks are performed.

If any validation check fails, the request termination is canceled and the accept phase continues with the next activation item.

The next activation item is determined as follows:

- If the failure is due to a field validation clause, there is an implicit INVALID for the field that failed validation, which causes a POSITION IMMEDIATE TO CURRENT ITEM to occur.
- If the failure is due to a validation response, the INVALID and POSITION steps in the response determine the next item.

If all validation checks succeed, the Form Manager resets the contents of the CURRENTITEM, FUNCTIONNAME, and FIELDIMAGE built-in form data items to spaces and terminates the accept phase.

4.4.16. Altering the Order of Activation Item Processing

Usually, the Form Manager processes activation items as indicated by POSITION response steps. However, some conditions can cause this general order of activation item processing to be modified.

The Form Manager performs special processing if one of the following is true:

- A function response does not specify a POSITION response step.

If a function response does not specify or imply a POSITION response step, the Form Manager begins processing the current activation item again. When this occurs, the Form Manager returns to the operator input stage. The Form Manager continues processing this activation item until the operator enters a function that contains a POSITION response step.

- A function response contains a POSITION IMMEDIATE or RETURN IMMEDIATE response step.

  When the Form Manager encounters a POSITION IMMEDIATE or RETURN IMMEDIATE response step, it skips the validation stages for the current activation item. However, the Form Manager does perform any field, group, icon, or panel exit responses.

  Only after the exit responses have been performed does the Form Manager begin processing the next current activation item (POSITION IMMEDIATE) or return control to your application program (RETURN IMMEDIATE).

- The Form Manager encounters a RETURN or EXIT HELP response step.

  When the Form Manager encounters a RETURN response step, it returns control to your application program after completing validation successfully. The Form Manager stops processing the current activation list, even though it may not have processed some activation items on the current activation list.

  The Form Manager continues processing the current activation item as normal. If the item associated with the current activation item fails validation, the Form Manager begins the operator input stage of accept processing again. When the item is validated, the Form Manager begins the termination check stage of accept phase processing. Once the termination check stage completes, the Form Manager terminates request processing and returns control to your application program.

- The Form Manager cannot validate input because of an operator entry error or an INVALID response step.

  If the Form Manager cannot complete processing because of an operator input error or an INVALID response step, it returns to the operator input stage of activation item processing and continues processing the current activation item.

- The Form Manager encounters a DEACTIVATE response step.

  If it encounters the DEACTIVATE response step, the Form Manager alters the activation list. The DEACTIVATE response step removes activation items from the activation list.

  If the activation item being removed is not the current activation item, the Form Manager immediately removes it from the activation list.

  If the activation item that is removed is the current activation item, the Form Manager completes the processing of that activation item by processing any field, group, icon, or panel exit responses. Only after the applicable exit responses have been performed does the Form Manager remove the activation item from the activation list.

- The PROTECTED WHEN attribute of an item has become true.

  If an item becomes protected because the PROTECTED WHEN attribute for that item is true, the Form Manager cannot process that activation item until it is unprotected. Therefore, it skips the activation item that is the protected item. The Form Manager determines which activation item to process next by searching the activation list for an item that is not protected.

When the Form Manager looks through the activation list for an activation item that is eligible for input, it begins with the activation item immediately following the one it could not process. If that activation item is eligible for input, the Form Manager processes that item. Otherwise, it searches forward through the activation list until it encounters either an activation item that is eligible for input or the end of the activation list.

If the Form Manager reaches the end of the activation list without finding an activation item that is eligible for input, it begins to look backward in the list. Starting from the point at which it began its forward search, the Form Manager searches backward until it encounters either an activation item that is eligible for input or the beginning of the activation list.

If the Form Manager reaches the beginning of the activation list without finding an activation item that is eligible for input, it processes any field, group, icon, or panel exit responses for the current activation item, and terminates accept phase processing. The Form Manager terminates accept phase processing by performing the termination check stage of activation item processing.

The activation item that is a protected item is not removed from the activation list. It may become unprotected, and therefore eligible for input, later during activation list processing. If the Form Manager encounters the activation item after the item is unprotected, it processes the activation item.

4.4.17. Help Processing

DECforms software provides you with two ways to provide help for your users. Both ways use response steps and the activation list. The first, and simpler way of specifying help, is to use the DECforms defaults. The second method is to use customized help.

If you use the DECforms defaults, you can specify two levels of help by declaring help messages and help panels in your form that the Form Manager displays at specified times during form processing. These declarations do not apply to PRINTER layouts.

You can declare help messages on panels, fields, groups, icons, and layouts within your form by using the USE HELP message clause. When you declare a USE HELP message clause within your form, the clause that you specify is displayed in the message panel when a MESSAGE HELP response step is executed.

By default, a MESSAGE HELP response step is executed when the user presses the function key defined as the Help key. (Different keys can be specified as Help keys. Depending on which terminal you have, DECforms software provides a default key as the Help key. For more information, see the appendix section of the *VSI DECforms IFDL Reference Manual*.)

The help message is the first level of help. You can also specify a help panel for a form element, which gives the user another level of assistance after the help message has been displayed. You specify a help panel on a form element by declaring the USE HELP PANEL clause within a field, group, icon, layout, or panel declaration, and by declaring a help panel.

In a HELP PANEL declaration, you can specify the viewport in which the help panel is to be displayed, the display attributes for the help panel, and the responses that occur when functions are invoked. You can also specify a help message for the help panel. You cannot specify a USE HELP PANEL clause within a HELP PANEL declaration. For more information on how to declare a help panel, see the *VSI DECforms IFDL Reference Manual*

4.4.17.1. Starting Help Processing

With the DECforms defaults, when the user presses the Help key the first time, the Form Manager starts help processing by executing the NEXT HELP built-in function response. The NEXT HELP

built-in function response specifies that if a help message is available for this form element, execute the MESSAGE HELP response step. The MESSAGE HELP response step displays the help message (the message clause you specified in your form as the USE HELP message clause) in the message panel.

The next time a user presses the Help key, the NEXT HELP built-in function response executes, and because the help message has already been displayed, an ENTER HELP response step is executed. The ENTER HELP response step sets the elementary condition HELP ACTIVE to true and creates a help activation list.

If after exiting help the user presses the Help key a third time, the Form Manager displays a "No help available" message.

4.4.17.2. How the Help Activation List Works

The help activation list is similar in some ways to the main activation list. The help activation list is created when an ENTER HELP response step is executed and is deleted when help is exited.

When the Form Manager processes the ENTER HELP response step, it checks to see if a help panel exists for the current help activation item. The Form Manager starts searching up the form hierarchy for help.

If the current help activation item is a field or icon and you have help specified at the panel level, the Form Manager displays that higher level help, unless the form item is explicitly declared as having NO HELP.

If NO HELP is omitted, the Form Manager continues searching upward until it reaches the top level of the hierarchy. If it reaches the top level and finds no help, the Form Manager displays the message, "No help available".

If the help panel has no fields, or if all the fields on the help panel are protected, the Form Manager will either activate a wait on the help panel or display the help panel depending on the display device.

If there are any unprotected items on the help panel, the Form Manager activates the panel, and then activates the items in the order specified in the PANEL declaration.

4.4.17.3. Using DECforms Defaults

Suppose you have a form that contains panel ACCOUNT. Field WITHDRAW is on panel ACCOUNT. There is a help panel associated with WITHDRAW and another associated with ACCOUNT. When the user asks for help on Field WITHDRAW, he gets a help message, and then the help panel associated with WITHDRAW. If he presses the Help key a third time, the Form Manager displays the message, "No help available".

If there is no help specified for field WITHDRAW, when the operator presses the Help key, the Form Manager checks to see if help is declared at this level in the form. The Form Manager finds no help message for WITHDRAW. The Form Manager begins searching up the form hierarchy for help, and when it encounters the help panel for panel ACCOUNT, it displays that help panel.

After the Form Manager displays the panel ACCOUNT help panel, it returns you to the place in the form where you were before help processing began.

4.4.17.4. Customized Help

If the default method of help processing does not provide you with the help you want, use response steps, elementary conditions, the main activation list, and the help activation list to create your own help processing. Appendix A provides information on help conditions.

The *VSI DECforms IFDL Reference Manual* describes built-in help responses.

The following example illustrates customized help processing.

```
Field DATA_TYPE                                          ❶
    Line 4
    Column 18
    Output Picture X(50)
    Use Help Message                                     ❷
        "Enter the VAX data type for the field's data" -
        " item.  Press HELP for a list of valid data " -
        " types or SELECT to choose from a list."
    Use Help Panel                                       ❸
        CREATE_FIELD_HELP_DT_PANEL
End Field
.
.
.
Help Panel CREATE_FIELD_HELP_DT_PANEL                    ❹
    Viewport HELP_VP                                     ❺
    Display
        %Keypad_Application
    Entry Response                                       ❻
        Activate                                         ❼
            Panel CREATE_FIELD_HELP_PANEL                ❽
            Panel CREATE_FIELD_HELP_PIC_PANEL            ❾
            Panel FUNCTION_KEY_HELP_PANEL                ❿
        Position To Icon DISMISS_ICON On                 ⓫
            CREATE_FIELD_HELP_DT_PANEL
    End Response
```

❶ Picture field DATA_TYPE is declared.
❷ A help message for field DATA_TYPE is specified in the Use Help Message.
❸ Use Help Panel specifies a help panel, CREATE_FIELD_HELP_DT_PANEL.
❹ Panel CREATE_FIELD_HELP_DT_PANEL is declared.
❺ CREATE_FIELD_HELP_DT_PANEL is displayed in Viewport HELP_VP. Up to this point, this is standard DECforms help processing.
❻ An entry response within CREATE_FIELD_HELP_DT_PANEL is declared.
❼ An ACTIVATE response step is declared, specifying additional panels to be activated on the help activation list while help is active.
❽ CREATE_FIELD_HELP_PANEL is activated.
❾ CREATE_FIELD_HELP_PIC_PANEL is activated.
❿ FUNCTION_KEY_HELP_PANEL is activated. When these help panels are activated, they are placed on the help activation list. Within each help panel, you can specify other responses, if desired.
⓫ Position To Icon DISMISS_ICON on CREATE_FIELD_HELP_DT_PANEL returns you to an icon in the help panel.

In the previous example, entry responses were used to activate additional items on the help activation list other than those declared on the initial help panel. While the help activation list is active, all ACTIVATE, DEACTIVATE, and POSITION response steps take place on the help activation list.

To customize your form's help processing, you must begin with a response, from which you can specify response steps. Within the response steps, you can check the elementary conditions that directly affect help.

These conditions are:

- HELP ACTIVE
- HELP MESSAGE EXISTS
- HELP PANEL EXISTS
- HELP MESSAGE AVAILABLE

For more information on responses and response steps, see the *VSI DECforms IFDL Reference Manual*. For more information on elementary conditions, see Appendix A.

4.5. Request Exit Response Phase

During this phase of processing, the Form Manager executes the exit response declared as a REQUEST EXIT RESPONSE in the external response for the current request.

A REQUEST EXIT RESPONSE is useful when request completion activities need to be done in the form. For example, a REQUEST EXIT RESPONSE can reset the values of certain form data items that are used within the request, so that they are in a known state when the next request is issued.

4.6. Form Data Collection Phase

The sixth phase of request processing for TRANSCEIVE requests, and the fifth phase for RECEIVE requests, is the form data collection phase. (The Form Manager performs this phase only for RECEIVE and TRANSCEIVE requests.) During this phase, the Form Manager moves data from form data items to record fields. (You can think of this phase as being the opposite of the data distribution phase explained in Section 4.2.)

For a receive record, the Form Manager supplies values for record fields with no associated form data: blanks for alphanumeric record fields and zeros for numeric record fields.

If the data type of a form data item does not match the data type of its corresponding record field, the Form Manager must convert the data in the form data item to the data type of the record field.

If the data in the form data items cannot be converted to the data type of the record fields, the Form Manager terminates the request and returns an error to your application program. If a fatal data conversion error occurs, the Form Manager does not alter the data in the record field in which the error occurred.

4.6.1. How Form Data Is Collected

When the Form Manager collects form data, it moves the values from the form data items into the application program record fields. During data collection, any record field or record field array element that does not receive data from a form data item or form data array element is set to a default.

Record fields and form record field array elements that are defined to contain alphanumeric data are set to spaces by default during data collection.

Record fields and record field array elements that are defined to contain numeric data are set to zeros during data collection. Record fields and record field array elements that are defined to contain date/time data are set to appropriate fields from midnight on November 17, 1858, the Smithsonian base date.

Note

The fields in the form record must have the same data type, length, and dimension as the corresponding fields in the application program record.

4.6.2. Data Conversion

If the data type of a form data item does not match the data type of its corresponding program record field, the Form Manager must convert the data in the form data item to the data type of the program record field.

If the Form Manager cannot do this, it terminates request processing and returns an error to your application program.

4.6.3. Using the TRANSFER Clause

Use the TRANSFER clause to specify that the name of a form data item should be treated as if it had a different name (which must be the same as a record field in a form record). The TRANSFER clause acts as a bridge between convenient names for record fields and convenient names for form data items.

The TRANSFER clause has other special uses. The SOURCE clause declares that the value of the record field is to be set from a form data item when the application program requests the given form record. The form data item can then broadcast its value to several record fields in the same form record. A record can have SOURCE and DESTINATION clauses pointing to different form data items to achieve special effects.

You can also use the TRANSFER clause to move data and record field arrays. For more information on the TRANSFER clause, see the *VSI DECforms IFDL Reference Manual*.

4.6.4. Shadow Records

In addition to the data contained within a form record, applications can optionally pass additional information that is associated with and describes special attributes of the record fields of the form record. When this information is used, it is passed in a shadow record whose structure is directly related to the form record.

The special information is of two varieties, depending on the direction of the data transfer. The application can get information about the modified status of the record fields returned in a receive record and it can supply information as to whether the last known values for modified field information should be updated.

Modified Fields

The Form Manager tracks the modified status of form data and returns this data to the application program if the program requests it in a shadow record. The application program may find this data useful in deciding whether certain actions must take place based on the data returned from the form.

Because keeping track of the modified status of form data might degrade performance in most implementations, and because you may not need to know the modified status of all form data, you can track form data items individually, or track all the form data items in a data group, or track all the form data items in a form. The Form Manager saves the last known values of tracked form data items only.

Modified data is associated with form data, which is directly connected to the record fields that are sent from and received into the program. What modified data means depends upon the last known value of

a form data item. The program has an expectation of the value of a form data item: the program knows what it has sent out to the form or what it received back from the form in a previous request. If the value being returned to the program in a receive record is different from this last known value, the value is marked as modified.

Modified data is maintained between requests. The program can send out a record field in a send request and later ask for that record field back in a receive request (perhaps several requests later). If the value of the form data item that is associated with that record field returned has changed because of any form activity (operator input, form procedural statements, data transfer) the returned data states that the record field has changed. If the value changed, the program receives data that the record field has not changed, because it is associated with the last known value of the associated form data item.

4.6.5. Receive Shadow Records

You may wish to track modified data in your application. During form data collection, the receive shadow record can be used to track such data.

The receive shadow record consists entirely of characters, and its length is 1 plus the number of record fields that the form record contains. The first character of the shadow record specifies the modified status of the entire record. If any record field in the record has been modified, the first character indicates that the record has been modified; if no record field has been modified, the first character indicates that the record has not been modified.

After the first character, the shadow record contains a single character for every record field in the record being returned. The correspondence between the shadow record and the received record is by ordinal position of record fields in the received record and ordinal position of characters in the shadow record, with an offset of 1, because of the additional character at the start of the shadow record.

A character returned in the field part of a shadow record is either 0 (record field not modified), X (the form did not request that modified data be kept for the record field, so the Form Manager has no status to report), or 1 (record field modified). If all field characters are 0, the first character of the shadow record (status for the entire record) is 0. If all field characters are either 0 or X, the first character is X. If any field character is 1, the first character is 1.

For example:

```
FORM DATA
    B CHARACTER(15) TRACKED
    C INTEGER  (1)  TRACKED
    D INTEGER  (2)  TRACKED
END DATA

FORM RECORD ELVIS
    B CHARACTER(15)
    C INTEGER  (1)
    D INTEGER  (2)
END RECORD
```

The record ELVIS is 18 bytes long. Its shadow record is 4 bytes: 1 for the entire record and 1 for each of the record fields.

```
FORM DATA TRACKED
    A INTEGER(4)
    GROUP G1 OCCURS 5
        GROUP G2 OCCURS 3
            D INTEGER(2)
```

```
            END GROUP
        END GROUP
        B CHARACTER(15)
    END DATA

    FORM RECORD ELVIS2
        A INTEGER(4) IMPLICIT SIGN
        GROUP G1 OCCURS 5
            GROUP G2 OCCURS 3
                D INTEGER(2)
            END GROUP
        END GROUP
        B CHARACTER(15)
    END RECORD
```

The record ELVIS2 has 49 data bytes. Its shadow record has 20 bytes: 1 for the entire record, 1 for A, 15 for the two-dimensional array G1.G2.D, 1 for B, and 2 for the sign character.

To see if record fields have been changed, a program can look at particular parts of a record by defining an appropriate shadow record structure in the program and asking the appropriate question on that structure.

For example, in COBOL:

```
    01 ELVIS2-SHADOW
       03 A             PIC X.
       03 G1            OCCURS 5.
          05 G2         OCCURS 3.
             07 D       PIC X.
       03 B             PIC X.
```

In the COBOL procedure section, the programmer could then write:

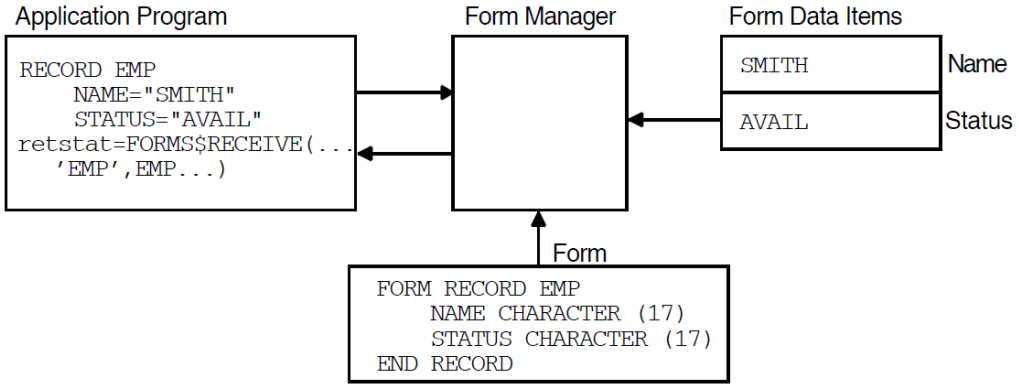
```
    IF D(3,2) OF ELVIS2-SHADOW = "1" THEN

change code for element D(3,2).

    IF G1(3) OF ELVIS2-SHADOW NOT = "000" THEN

change code for row 3.
```

For some requests, the program might not affect the last known value of form data, especially in a program that is an escape routine making recursive requests on the Form Manager. Values passed to and from the form by the escape routine can effect form data, which would cause the last known value to be updated. If you choose not to change the last known value in the OpenVMS API, the escape routine can run transparently to the main requester.

If no receive shadow record is supplied on a request, the Form Manager does not assume that the program knows about the change in a form data item's value; specifically, the Form Manager does not update the last known value in the form for the record fields in a receive record unless it also passes a receive shadow record.

Similarly, a request may specify, in the first character of a send shadow record, that it does not want the values in the receive record to update the last known values in the form. If an *N* (either uppercase or lowercase) is specified in the first byte of the shadow record, the Form Manager does not update the last known value; if any other character is specified in the first byte, the last known value is updated. Figure 4.2 shows the exchange of information that can occur during a RECEIVE request.

Figure 4.2. Function of the Form Manager During Data Collection



The following steps describe the process outlined in Figure 4.2:

1. The RECEIVE request asks that data stored in the form data items that correspond to the EMP form record be returned to the application program.

2. The Form Manager looks in the form for a form record EMP, and sees that EMP is composed of the form record fields NAME and STATUS.

3. The Form Manager moves the values in the form data items NAME and STATUS to the NAME and STATUS application program record fields. If necessary, the Form Manager converts the data in the form data items to the data type of the record fields.

4.6.6. Data Transfer of Arrays

When data transfer takes place during data distribution and data collection; and the record fields, or form data items, or both are arrays; the amount of data that is transferred depends on the dimensions of the record fields and form data items involved.

The following rules explain how data is transferred between a record field and a form data item during data distribution and data collection, when one or both are arrays:

- When either the record field or the form data item that takes part in a data transfer is a whole array (as opposed to an array element), only a single data value is transferred between the record field and the form data item. The Form Manager uses the lowest subscripted member of the array in the transfer. The remaining members of the array are not transferred.

  For example, if record field G1.A is not an array and it transfers to or from form data item G1.A, which is a one-based array that occurs five times, only element G1.A (1) of the form data array is transferred to and from record field G1.A. The other four elements of the form data array G1.A are not transferred.

- When both the record field and the form data item are arrays of the same dimension and contain the same number of array elements in each dimension, elements are transferred directly from one array to the other. The element with the lowest subscript of each array is transferred first, the element with the second lowest subscript is transferred second, and so forth.

- When both the record field and the form data item arrays are two-dimensional but the size of the two arrays is different (the OCCURS clauses specify a different number of occurrences), the number of elements that are transferred is the minimum of the number of occurrences in each dimension for each array, independent of all other dimensions.

Given that DECforms arrays are always in row major order, the Form Manager transfers rows of the record field and form data arrays as if they were one-dimensional arrays:

1. The Form Manager transfers the first row of the record field and form data arrays until one of the arrays has no more elements in its row.

2. The Form Manager transfers the second row of each array until one of the arrays has no more elements in its second row.

3. The Form Manager continues transferring rows of the record field and form data arrays in this manner until one of the arrays has no more rows to transfer.

If the form data array is shorter than the form record array, the remainder is filled with default characters.

- When one of the arrays (either the record field array or the form data array) is one-dimensional and the other is two-dimensional, the Form Manager treats the one-dimensional array as a two-dimensional array, with the first dimension being of size 1 (as if OCCURS 1 were specified for the outermost group of two nested, multiply occurring groups). The Form Manager proceeds as if both arrays are two-dimensional but the size of the two arrays is different.

  For example, if record field G1.A is a one-dimensional array and form data item G1.G2.A is a two-dimensional array, the Form Manager treats the first row of the form data array G1.G2.A as a one-dimensional array and proceeds as if both are one-dimensional arrays. The transfer is made to or from only one row of the two-dimensional form data array G1.G2.A, with no wraparound to the beginning of the next row.

  Because of the way the Form Manager performs data transfer of arrays, not all elements of a record field array may actually be transferred to or from a form data array during data distribution or data collection.

4.7. Request Termination Phase

The last phase of external request processing for all requests is the terminate request phase. The Form Manager performs this phase of processing for each external request that you call, although the Form Manager performs special termination processing on a DISABLE call. This phase completes the request and returns control to your application program.

The Form Manager returns a value to the return status variable for the external request. This value is the OpenVMS condition value of the most severe error that the Form Manager encountered during processing in the OpenVMS API. In the portable API, the Forms Manager returns a FIMS error number representing the most severe error.

The Form Manager performs special termination processing on a DISABLE call. First, the Form Manager unloads the form. After it unloads the form, the Form Manager detaches the display device. The session is then terminated, along with the existing form data.

If you passed receive control text in your request call, the Form Manager also returns other exception conditions and informational messages in this receive control text. The receive control text is composed of individual **receive control text items**, each of which is five characters.

The number of receive control text items that a receive control message contains depends on the length of the receive control text and the number of exceptions encountered and messages that must

be returned. In the portable API, the receive control message text must be 25 bytes long, while in the OpenVMS API, the length of the text is user-specified. Appendix B lists the receive control text items.

You can define application-specific receive control text items in a RETURN response step. For information on defining receive control text items, see the RESPONSE STEP syntax in the *VSI DECforms IFDL Reference Manual*.

Receive control text items have the following defined format:

[WSXXX]

[W]

W indicates whether an exception occurred. The letter “E ” in this position indicates that an exception occurred. A space indicates that no exception occurred; the control text item is informational.

[S]

The letter S in the second position indicates the source of the receive control text item depending on which character appears in this position. The characters that can appear are as follows:

Character	**Receive Control Text Item Source**
F	Indicates that the form designer defined the receive control text item in the form.
I	Indicates that the receive control text item was defined by DECforms software.
S	Indicates that the receive control text item was defined by the FIMS standard.

[XXX]

The XXX designation contains the control code as specified in Appendix B.

For example, in the receive control text ES004, the E signals an exception that was defined by the FIMS standard (S); the 004 control code indicates that the Form Manager could not establish a session with *session-id* specified in the argument of an ENABLE request. Appendix B lists this and other control codes in numeric order.

Chapter 5. Using the OpenVMS API

DECforms software supports both an OpenVMS application programming interface (API), and a portable API that supports C and FORTRAN bindings. This chapter describes each DECforms request for users of the OpenVMS API. For similar information about the portable API, see Chapter 6.

Each DECforms request is described in a structured format that provides the following information:

- The name of the request and a sentence describing its purpose.
- The format of the request. Optional arguments are enclosed in square brackets ([ ]).
- A list that provides information on the return status variable, which describes the information returned by the program to the Form Manager. This information includes what data type the return status variable should be, the access that the Form Manager needs to the return status variable, and the passing mechanism of the return status variable.
- A description of each argument. A list at the beginning of each argument description indicates what the data type of the argument should be, the access that the Form Manager needs to the argument, and the passing mechanism of the argument.
- A complete description of the request.
- The possible return values for the request, listed in alphabetical order.
- One or more examples of using each request in the FORTRAN programming language.

The Form Manager ignores case in arguments that you pass. For example, the Form Manager considers the name RECORD_ONE to be equivalent to the name record_one.

For examples of how to include requests in an application program, see Chapter 2. For information on how the Form Manager processes requests, see Chapter 4.

5.1. DECforms Requests

ENABLE

ENABLE — Initializes a DECforms session.

Format

```
ret-status=FORMS$ENABLE

            (
             form-object-address,
             display-device-specification,
             session-id,
             [file-specification,]
             [form-specification, receive-control-text,
```

```
 receive-control-text-count],
[send-control-text, send-control-text-count],
[timeout],
[parent-request-id],
[request-options]
)
```

Returns

OpenVMS usage:	cond_value
type:	longword (unsigned)
access:	write only
mechanism:	by value

Arguments

form-object-address

OpenVMS usage:	longword (unsigned)
type:	longword (unsigned)
access:	read only
mechanism:	by value

Required argument that is either the FORMS$AR_FORM_TABLE symbol name or 0. The FORMS $AR_FORM_TABLE symbol is defined in an object module stored in a system library. This symbol is a pointer to the address of a special object module, called a form object.

The form object module contains the addresses of escape routines and forms that are linked with the application.

If no form object module is needed for the session, you can pass 0 in *form-object-address*.

display-device-specification

OpenVMS usage:	char_string
type:	character-coded text string
access:	read only
mechanism:	by descriptor

Required argument that is a character-string name for the device to be used as the display device during this session.

You can define a logical name for the device name and pass that logical name in *display-device-specification*.

session-id

OpenVMS usage:	char_string

type:	character-coded text string
access:	write only
mechanism:	by descriptor

Required argument that is filled by the Form Manager; the Form Manager returns a 16-byte character session identification string.

Session-id associates the display device with the form that was loaded into memory. You must pass this session identification string in subsequent request calls to tell the Form Manager which display device and form you want the Form Manager to use.

The string you specify must be 16 bytes long.

file-specification

OpenVMS usage:	char_string
type:	character-coded text string
access:	read only
mechanism:	by descriptor

Optional argument that specifies the character-string file specification of the form file to be used during this session. The file specification can include node name, device name, directory specification, file name, file type, and file version number.

If you omit the node name, device name, or directory specification, the Form Manager uses the node you are logged in to, the current default device, and the current default directory, respectively.

If you omit the file type, the Form Manager looks for a file with the .form or .exe type. If you omit the file version number, the Form Manager uses the latest version (highest version number) of the specified file.

You can define a logical name for the file specification and pass that logical name in *file-specification*.

If you do not specify this argument, you must specify *form-specification*. If your form is linked into a shareable image, you must specify this argument and *form-specification*.

The following table shows what you must specify for a form name or a form file when specifying *file-specification*. The X designates what you must supply.

	.FORM	**.EXE**	**Linked In**
file-specification	X	X	
form-specification		X	X

form-specification

OpenVMS usage:	char_string
type:	character-coded text string
access:	read only

mechanism:	by descriptor

Optional argument that specifies the name of the form in the IFDL source file. The Form Manager first looks for a .form file, then searches for a *file-specification*.

If you do not specify this argument, you must specify *file-specification*. If the form specified in *file-specification* is a .form file, *form-specification* is ignored. If your form is linked into a shareable image, you must specify this argument and the *file-specification*.

receive-control-text

OpenVMS usage:	char_string
type:	character-coded text string
access:	write only
mechanism:	by descriptor

Optional argument that returns status information from the Form Manager to the application program when the request is terminated. The status information is divided into a maximum of five receive control text items.

Each receive control text item is five single-byte characters long, so the variable you pass in this argument must be able to store a string that is a multiple of five characters long. If you pass this argument, you must also pass *receive-control-text-count*.

If the length specified in the OpenVMS descriptor for *receive-control-text* is too short to hold all the receive control text items that the Form Manager attempts to return, the Form Manager discards those receive control text items that do not fit.

receive-control-text-count

OpenVMS usage:	longword_unsigned
type:	longword (unsigned)
access:	write only
mechanism:	by reference

Optional argument that specifies the number of receive control text items returned by the Form Manager (each of which is five characters long) in the receive control text. This number indicates the number of receive control text items returned by the Form Manager to the application.

When the Form Manager terminates the request, it stores the number of receive control text items that it is returning in this argument. This argument is set to 0 at the start of each request. If you pass *receive-control-text*, you must pass this argument.

send-control-text

OpenVMS usage:	char_string
type:	character-coded text string
access:	read only
mechanism:	by descriptor

Optional argument in which you can pass zero to five send control text items to the Form Manager from the application program. Each send control text item must be one to five single-byte characters long and should name a control text response stored in the form. If you pass this argument, you must also pass the *send-control-text-count* argument.

You can pass a send control text item that is less than five characters long. However, if you are specifying an additional send control text item, the Form Manager requires you to pad the send control text with blanks to five 1-byte characters.

If you omit this argument, the Form Manager will not execute any control text response. If a send control text item specifies a control text response that does not exist, the Form Manager either processes the next send control text item or proceeds to the next request phase.

send-control-text-count

OpenVMS usage:	longword_unsigned
type:	longword (unsigned)
access:	read only
mechanism:	by reference

Optional argument in which you specify the number of control text items in *send-control-text*. You can specify zero to five control text items in this argument. If you pass *send-control-text*, you must pass this argument.

timeout

OpenVMS usage:	word_unsigned
type:	word (unsigned)
access:	read only
mechanism:	by reference

Optional argument in which you specify a timeout value for the entry timer. The entry timer controls the maximum number of seconds that can elapse between each operator keystroke.

The timeout value you pass in this argument supersedes any timeout value declared in the form.

If you omit this argument, the operator has unlimited time between each keystroke, unless you specify a timeout value in the form. If the timeout value is set to 0, the operator has unlimited time between each keystroke.

parent-request-id

OpenVMS usage:	char_string
type:	character-coded text string
access:	read only
mechanism:	by descriptor

Optional argument specifying the **parent request** of the new request. When a request that is currently executing invokes an escape routine that invokes another DECforms request, *parent-request-id* specifies the original request that called the escape routine.

This argument is intended only for using DECforms calls from within an escape routine. There can be no *parent-request-id* argument for a request called from a nonprocedural escape routine called from within the application.

You can make *parent-request-id* available within an escape routine by passing the built-in form data item PARENTREQUESTID as one of the arguments to the escape routine within the CALL response step.

This argument is 24 bytes long.

Note

When an escape routine calls the current session or another session, the calling request hangs unless:

1. A built-in form data item of PARENTREQUESTID is declared as a form data item declaration.
2. The declared form data item is passed as a descriptor to the escape routine.
3. The escape routine passes the descriptor as a parameter to the request.

Once the built-in form data item declaration is declared and the PARENTREQUESTID is passed to the escape routine, the DECforms request being called from the escape routine is processed while the parent request completes.

request-options

OpenVMS usage:	item_list_2
type:	longword (unsigned)
access:	read only
mechanism:	by reference

Optional item list containing request-specific arguments that control the environment of the request.

Request-options is the address of an OpenVMS item list containing zero or more request arguments. Each item consists of three longwords.

The first longword consists of two fields: a 1-word field that contains the length of the application buffer, and a 1-word field containing the item code value.

The second longword contains the address of the application buffer that either contains or will receive the data.

The third longword contains the address of a word to receive the length of the data written into the buffer. (The third longword is optional and is useful when the specified item code indicates that data will be returned to the application.)

The item list is terminated by a longword containing a 0.

Certain item codes correspond to logical names that the Form Manager translates during request processing. If you specify an item code, the Form Manager reads the item code directly and does not attempt to translate the corresponding logical names.

Table 5.1 contains the item codes that can be specified and their explanations.

Table 5.1. FORMS$ENABLE Request Options

Item Code	Description
FORMS$K_ASTADR	The asynchronous system trap (AST) service routine to be executed when the request is complete. The buffer address field of the item contains the address of the entry mask for this routine.
FORMS$K_ASTPRM	The AST argument to be passed to the AST service routine. The buffer address field of the item contains the address of a longword value containing the value to be passed.
FORMS$K_EFN	Event flag the Form Manager sets when it completes the request. The buffer address field of the item contains the event flag number. The Form Manager clears the specified event flag at the beginning of the request.
FORMS$K_IMAGE	File specification of a shareable image that is searched for procedural escapes. This item code can appear up to eight times in the item list: the order of appearance specifies the order in which the images will be searched for procedural escapes. The buffer address field of the item contains the address of a string that contains the image file specification.
FORMS$K_LANGUAGE	Natural language to use when selecting a layout to enable. The buffer address field of the item contains the address of a string that contains a language type. The string attempts to match the language description in the foreign layout. If it does not find a match, it selects the next best fit.
FORMS$K_PRINTFILE	File specification of the file to be printed using the PRINT response step. The buffer address field of the item contains the address of a string that contains the print file specification.
FORMS$K_RSB	The quadword request status block to receive the completion status of the request. The buffer address field of the item contains the address of a DECforms quadword status block where the Form Manager writes the return status of the request. The Form Manager returns the final condition value of the request in the first longword of the status block. The second longword is reserved for future use.
FORMS$K_TRACE	Tracing indicator for the session. The buffer address field of the item can be specified as zero or any nonzero value. A value of zero indicates that

Item Code	Description
	trace is disabled. Any nonzero value indicates that trace is enabled.
FORMS$K_TRACEFILE	File specification of the file to be used to trace the execution of the session. The buffer address field of the item contains the address of a string that contains the trace file specification. If another request on this session opens the trace file, this item code is ignored.

Description

The ENABLE request loads the specified form file into memory, selects an appropriate layout, and attaches the specified display device. The Form Manager also returns a unique session-identification string and initializes all form data items. For more information about initializing an ENABLE request, see Section 4.1.2.

You can include more than one ENABLE request call in your application program (the Form Manager supports multiple synchronous requests to display devices). For example, the operator's terminal can be enabled for two different sessions. The Form Manager can perform operations in any enabled session.

You cannot enable an executable form image located on a remote node due to a restriction in LIB $FIND_IMAGE_SYMBOL.

Return Values

FORMS$_BADARG

Bad argument or incorrect argument format for the request.

FORMS$_CANCELLED

Request processing interrupted by arrival of CANCEL request.

FORMS$_CANTOPENDIC

The Form Manager could not open the Kana to Kanji dictionary.

FORMS$_INVDEVICE

The device specified on the ENABLE call was invalid.

FORMS$_LOADFORM

The Form Manager could not load the specified form.

FORMS$_NOLAYOUT

No layout in this form fits the terminal type, language, and display size.

FORMS$_NOLICENSE

No DECforms software license is active.

FORMS$_NORMAL

FORMS$ENABLE was successfully completed.

FORMS$_OPENFORM

The Form Manager could not open the form file for input.

FORMS$_READFORM

The Form Manager could not read the form file.

FORMS$_TIMEOUT

Input was not completed in the time specified.

Examples

1.
```
CHARACTER*16 session_id
INTEGER forms_status ! check status

    .
    .
    .
forms_status = forms$enable (  FORMS$AR_FORM_TABLE, ! form object
                                                       address
1                              'sys$input:',        ! Device name
2                              session_id,          ! return from forms$
3                              'forms$sample_form') ! Name of form file
               CALL check_forms_status( forms_status )

    .
    .
    .
```

The request call in this example causes the Form Manager to perform the following tasks:

a. Enable form file on the device specified by SYS$INPUT.

b. Validate the arguments in the request.

c. Attach the display device named by SYS$INPUT.

d. Create a session-identification string and store it in *session_id*.

e. Load the forms$sample_form.form form file into memory from the current default device and directory.

f. Process the ENABLE response, if defined.

g. Return control to the application program.

2.
```
CHARACTER*16 session_id
INTEGER forms_status

    .
    .
```

```
    .
forms_status = forms$enable (   ,                ! form object address
1                               'sys$input:',    ! Device name
2                               session_id,      ! return from forms
                                                 ! $enable
3                               'my_form.form') ! Name of form file

    .
    .
    .
                CALL check_forms_status( forms_status )
```

By taking the following steps, the request call in this example causes the Form Manager to load a form file where there are no procedural escapes being called:

a. Validate the arguments in the request.

b. Attach the display device named by SYS$INPUT.

c. Create a session-identification string and store it in *session_id*.

d. Load the my_form.form form file into memory from the current default device and directory.

e. Process the ENABLE response, if defined.

f. Return control to the application program.

DISABLE

DISABLE — Terminates a session.

Format

```
ret-status = FORMS$DISABLE
            (
            session-id,
            [receive-control-text, receive-control-text-count],
            [send-control-text, send-control-text-count],
            [timeout],
            [parent-request-id],
            [request-options]
            )
```

Returns

OpenVMS usage:	cond_value
type:	longword (unsigned)
access:	write only
mechanism:	by value

Arguments

session-id

OpenVMS usage:	char_string
type:	character-coded text string
access:	read only
mechanism:	by descriptor

Required argument that contains a unique session-identification string. The Form Manager returns the session-identification string to your application program during the processing of an ENABLE request.

You must pass *session-id* because the session-identification string specifies which display device and session you want the Form Manager to use during this request.

The string you specify must be 16 bytes long.

receive-control-text

OpenVMS usage:	char_string
type:	character-coded text string
access:	write only
mechanism:	by descriptor

Optional argument that returns status information from the Form Manager to the application program when the request is terminated. The status information is divided into a maximum of five receive control text items.

Each receive control text item is five characters long, so the variable you pass in this argument must be able to store a string that is a multiple of five characters long. If you pass this argument, you must also pass the *receive-control-text-count* argument.

If the length specified in the OpenVMS descriptor for *receive-control-text* is too short to hold all the receive control text items that the Form Manager attempts to return, the Form Manager discards those receive control text items that do not fit.

receive-control-text-count

OpenVMS usage:	longword_unsigned
type:	longword (unsigned)
access:	write only
mechanism:	by reference

Optional argument that specifies the number of receive control text items returned by the Form Manager (each of which is five characters long) in the receive control text.

When the Form Manager terminates the request, it stores the number of receive control text items that it is returning in this argument. This argument is set to 0 at the start of each request. If you pass *receive-control-text*, you must pass this argument.

send-control-text

OpenVMS usage:	char_string
type:	character-coded text string
access:	read only

mechanism:	by descriptor

Optional argument in which you can pass zero to five send control text items to the Form Manager from the application program. Each send control text item must be one to five single-byte characters long and should name a control text response stored in the form. If you pass this argument, you must also pass *send-control-text-count*.

You can pass a send control text item that is less than five characters long. However, if you are specifying an additional send control text item, the Form Manager requires you to pad the send control text with blanks to five 1-byte characters.

If you omit this argument, the Form Manager will not execute any control text response. If a send control text item specifies a control text response that does not exist, the Form Manager either processes the next send control text item or proceeds to the next request phase.

send-control-text-count

OpenVMS usage:	longword_unsigned
type:	longword (unsigned)
access:	read only
mechanism:	by reference

Optional argument in which you specify the number of control text items in *send-control-text*. You can specify zero to five control text items in this argument. If you pass *send-control-text*, you must pass this argument.

timeout

OpenVMS usage:	word_unsigned
type:	word (unsigned)
access:	read only
mechanism:	by reference

Optional argument in which you specify a timeout value for the entry timer. The entry timer controls the maximum number of seconds that can elapse between each operator keystroke.

The timeout value you pass in this argument supersedes any timeout value declared in the form.

If you omit this argument, the operator has unlimited time between each keystroke, unless you specify a timeout value in the form. If the timeout value is set to 0, the operator has unlimited time between each keystroke.

parent-request-id

OpenVMS usage:	char_string
type:	character-coded text string
access:	read only
mechanism:	by descriptor

Optional argument specifying the parent request of the new request. When a request that is currently executing invokes an escape routine that invokes another DECforms request, *parent-request-id* specifies the original request that called the escape routine.

This argument is intended only for when you issue DECforms calls from within an escape routine. There can be no *parent-request-id* argument for a request called from a nonprocedural escape routine called from within the application.

You can make *parent-request-id* available within an escape routine by passing the built-in form data item PARENTREQUESTID as one of the arguments to the escape routine within the CALL response step.

This argument is 24 bytes long.

Note

When an escape routine calls the current session or another session, the calling request hangs unless:

1. A built-in form data item of PARENTREQUESTID is declared as a form data item declaration.
2. The declared form data item is passed as a descriptor to the escape routine.
3. The escape routine passes it as a parameter to the request.

Once the built-in form data item declaration is declared and the PARENTREQUESTID is passed to the escape routine, the DECforms request being called from the escape routine is processed while the parent request completes.

request-options

OpenVMS usage:	item_list_2
type:	longword (unsigned)
access:	read only
mechanism:	by reference

Optional item list containing request-specific arguments that control the environment of the request.

Request-options is the address of an OpenVMS item list containing zero or more request arguments. Each item consists of three longwords.

The first longword consists of two fields: a 1-word field that contains the length of the application buffer, and a 1-word field containing the item code value.

The second longword contains the address of the application buffer that either contains or will receive the data.

The third longword contains the address of a word to receive the length of the data written into the buffer. (The third longword is optional and is useful when the specified item code indicates that data will be returned to the application.)

The item list is terminated by a longword containing a 0.

Certain item codes correspond to logical names that the Form Manager translates during request processing. If you specify an item code, the Form Manager reads the item code directly and does not attempt to translate the corresponding logical names.

Table 5.2 shows the request options that you can specify.

Table 5.2. FORMS$DISABLE Request Options

Item Code	Description
FORMS$K_ASTADR	The asynchronous system trap (AST) service routine to be executed when the request is complete. The buffer address field of the item contains the address of the entry mask for this routine.
FORMS$K_ASTPRM	The AST argument to be passed to the AST service routine. The buffer address field of the item contains the address of a longword value containing the value to be passed.
FORMS$K_EFN	Event flag the Form Manager sets when it completes the request. The buffer address field of the item contains the event flag number. The Form Manager clears the specified event flag at the beginning of the request.
FORMS$K_NO_TERM_IO	Item code to permit a multiterminal application to disable its sessions without the risk that a terminal that is not accepting output will hang the shutdown process. If this item code is present and its value is nonzero, the disable request will not perform terminal I/O. Any output normally sent to the terminal during this request is discarded. The accept phase, which is the only phase during which input from the operator is accepted, is bypassed. If a session is disabled with NO_TERM_IO and another session remains open to that terminal, the other session can continue to be used normally. The first request made to the other session causes the screen to be refreshed if any terminal output from the first session was discarded.
FORMS$K_PRINTFILE	File specification of the file to be printed using the PRINT response step. The buffer address field of the item contains the address of a string that contains the print file specification.
FORMS$K_RSB	The quadword request status block to receive the completion status of the request. The buffer address field of the item contains the address of a DECforms quadword status block where the Form Manager writes the return status of the request. The Form Manager returns the final condition value of the request in the first longword of the status block. The second longword is reserved for future use.
FORMS$K_TRACE	Tracing indicator for the session. The buffer address field of the item can be specified as zero or

Item Code	Description
	any nonzero value. A value of zero indicates that trace is disabled. Any nonzero value indicates that trace is enabled.
FORMS$K_TRACEFILE	File specification of the file to be used to trace the execution of the session. The buffer address field of the item contains the address of a string that contains the trace file specification. If another request on this session opens the trace file, this item code is ignored.

Description

The DISABLE request terminates the specified session. The Form Manager detaches the display device and the form that are associated with the session-identification string. Once the Form Manager completes the processing of this request, the session specified by the session-identification string no longer exists. The values of all form data items in the form are disabled last.

Return Values

FORMS$_BADARG

Bad argument or incorrect argument format for request.

FORMS$_CANCELLED

Request processing interrupted by arrival of CANCEL request.

FORMS$_INUSE

The program attempted to disable a form that is in current use.

FORMS$_NORMAL

FORMS$DISABLE was successfully completed.

FORMS$_NOSESSION

The session-identification string passed in *session-id* does not match an existing session.

FORMS$_TIMEOUT

Input was not completed in the specified time.

Examples

1.
```
CHARACTER*16 session_id
INTEGER forms_status,
1       count/1/

      .
      .
      .
```

```
forms_status = FORMS$DISABLE( session_id )
CALL check_forms_status( forms_status )
```

The request call in this example causes the Form Manager to perform the following tasks:

a. Validate the arguments in the request.

b. Verify that *session_id* names a valid session.

c. Perform the DISABLE response, if defined.

d. Unload the form specified by *session_id*.

e. Detach the display device specified by *session_id*.

f. Return control to the application program.

2.
```
CHARACTER*16 session_id
CHARACTER*25 rec_ctrl_txt, send_ctrl_txt
INTEGER forms_status, reccount, sendcount, timeout

⋮
forms_status =
FORMS
$DISABLE(session_id,rec_ctrl_txt,reccount,send_ctrl_txt,sendcount,,,)
CALL check_forms_status( forms_status )
```

The request call in this example causes the Form Manager to perform the following tasks:

a. Validate the arguments in the request.

b. Verify that *session_id* names a valid session.

c. Perform the control text responses named in *send_ctrl_txt*.

d. Perform the DISABLE response, if defined.

e. Unload the form specified by *session_id*.

f. Detach the display device specified by *session_id*.

g. Store any receive control text items in *rec_ctrl_txt*.

h. Return control to the application program.

SEND

SEND — Copies data from an application program record to form data items.

Format

```
ret-status = FORMS$SEND
            (
            session-id,
            send-record-name,
```

```
send-record-count,
[receive-control-text, receive-control-text-count],
[send-control-text, send-control-text-count],
[timeout],
[parent-request-id],
[request-options],
[
[send-record-message],
[send-shadow-record],
] . . .
)
```

Returns

OpenVMS usage:	cond_value
type:	longword (unsigned)
access:	write only
mechanism:	by value

Arguments

session-id

OpenVMS usage:	char_string
type:	character-coded text string
access:	read only
mechanism:	by descriptor

Required argument that contains a unique session-identification string. The Form Manager returns the session-identification string to your application program during the processing of an ENABLE request.

You must pass *session-id* because the session-identification string identifies which DECforms session and form you want the Form Manager to use during this request.

The string you specify in this argument must be 16 bytes long.

send-record-name

OpenVMS usage:	char_string
type:	character-coded text string
access:	read only
mechanism:	by descriptor

Required argument that is a single-byte character string name of the form record (for single record transfers) or record list (for multiple record transfers) to which the Form Manager is passing data. *send-record-name* is used to search for and trigger a send response within the form file.

The character string you pass in this argument must match a record name, or record list name, in the form.

send-record-count

OpenVMS usage:	longword_unsigned
type:	longword (unsigned)
access:	read only
mechanism:	by reference

Required argument that contains the number of send record items passed to the form. (A send record item is made up of two parts: *send-record-message* and *send-shadow-record*. You cannot specify *send-shadow-record* unless you also specify *send-record-message*.) This number must also match the number of records in the form specified for *send-record-name*.

This value must be greater than or equal to 0. If *send-record-count* is 0, then *send-record-name* must correspond to an empty record definition in the IFDL.

receive-control-text

OpenVMS usage:	char_string
type:	character-coded text string
access:	write only
mechanism:	by descriptor

Optional argument that returns status information from the Form Manager to the application program when the request is terminated. The status information is divided into a maximum of five receive control text items.

Each receive control text item is five single-byte characters long, so the variable you pass in this argument must be able to store a string that is a multiple of five characters long. If you pass this argument, you must also pass *receive-control-text-count*.

If the length specified in the descriptor for *receive-control-text* is too short to hold all the receive control text items that the Form Manager attempts to return, the Form Manager discards those receive control text items that do not fit.

receive-control-text-count

OpenVMS usage:	longword_unsigned
type:	longword (unsigned)
access:	write only
mechanism:	by reference

Optional argument that specifies the number of receive control text items (each of which is five characters long) in the receive control message. This number indicates the number of receive control text items returned by the Form Manager to the application.

When the Form Manager terminates the request, it stores the number of receive control text items that it is returning in this argument. This argument is set to 0 at the start of each request. If you pass *receive-control-text*, you must pass this argument.

send-control-text

OpenVMS usage:	char_string

type:	character-coded text string
access:	read only
mechanism:	by descriptor

Optional argument in which you can pass zero to five send control text items to the Form Manager from the application program. Each send control text item must be one to five single-byte characters long and should name a control text response stored in the form. If you pass this argument, you must also pass the *send-control-text-count argument*.

You can pass a send control text item that is less than five characters long. However, if you are specifying an additional *send-control-text* item, the Form Manager requires you to pad *send-control-text* with blanks to five 1-byte characters.

If you omit this argument, the Form Manager will not execute any control text response. If a send control text item specifies a control text response that does not exist, the Form Manager either processes the next send control text item or proceeds to the next request phase.

send-control-text-count

OpenVMS usage:	longword_unsigned
type:	longword (unsigned)
access:	read only
mechanism:	by reference

Optional argument in which you specify the number of control text items in *send-control-text*. You can specify zero to five control text items in this argument. If you pass *send-control-text*, you must pass this argument.

timeout

OpenVMS usage:	word_unsigned
type:	word (unsigned)
access:	read only
mechanism:	by reference

Optional argument in which you specify a timeout value for the entry timer. The entry timer controls the maximum number of seconds that can elapse between each operator keystroke.

The timeout value you pass in this argument supersedes any timeout value declared in the form.

If you omit this argument, the operator has unlimited time between each keystroke, unless you specify a timeout value in the form. If the timeout value is set to 0, the operator has unlimited time between each keystroke.

parent-request-id

OpenVMS usage:	char_string
type:	character-coded text string
access:	read only

mechanism:	by descriptor

Optional argument specifying the parent request of the new request. When a request that is currently executing invokes an escape routine that involves another DECforms request, *parent-request-id* specifies the original request that called the escape routine.

This argument is intended only for when you send DECforms calls from within an escape routine. There can be no *parent-request-id* argument for a request called from a nonprocedural escape routine called from within the application.

You can make *parent-request-id* available within an escape routine by passing the built-in form data item PARENTREQUESTID as one of the arguments to the escape routine within the CALL response step.

This argument is 24 bytes long.

Note

When an escape routine calls the current session or another session, the calling request hangs unless:

1. A built-in form data item of PARENTREQUESTID is declared as a form data item declaration.
2. The declared form data item is passed as a descriptor to the escape routine.
3. The escape routine passes it as a parameter to the request.

Once the built-in form data item declaration is declared and the PARENTREQUESTID is passed to the escape routine, the DECforms request being called from the escape routine is processed while the parent request completes.

request-options

OpenVMS usage:	item_list_2
type:	longword (unsigned)
access:	read only
mechanism:	by reference

Optional item list containing request-specific arguments that control the environment of the request.

Request-options is the address of an OpenVMS item list containing zero or more request arguments. Each item consists of three longwords.

The first longword consists of two fields: a 1-word field that contains the length of the application buffer, and a 1-word field containing the item code value.

The second longword contains the address of the application buffer that either contains or will receive the data.

The third longword contains the address of a word to receive the length of the data written into the buffer. (The third longword is optional and is useful when the specified item code indicates that data will be returned to the application.)

The item list is terminated by a longword containing a 0.

Certain item codes correspond to logical names that the Form Manager translates during request processing. If you specify an item code, the Form Manager reads the item code directly and does not attempt to translate the corresponding logical names.

Table 5.3 contains the item codes that you can specify.

Table 5.3. FORMS$SEND Request Options

Item Code	Description
FORMS$K_ASTADR	The asynchronous system trap (AST) service routine to be executed when the request is complete. The buffer address field of the item contains the address of the entry mask for this routine.
FORMS$K_ASTPRM	The AST argument to be passed to the AST service routine. The buffer address field of the item contains the address of a longword value containing the value to be passed.
FORMS$K_EFN	Event flag the Form Manager sets when it completes the request. The buffer address field of the item contains the event flag number. The Form Manager clears the specified event flag at the beginning of the request.
FORMS$K_PRINTFILE	File specification of the file to be printed using the PRINT response step. The buffer address field of the item contains the address of a string that contains the print file specification.
FORMS$K_RSB	The quadword request status block to receive the completion status of the request. The buffer address field of the item contains the address of a DECforms quadword status block where the Form Manager writes the return status of the request. The Form Manager returns the final condition value of the request in the first longword of the status block. The second longword is reserved for future use.
FORMS$K_TRACE	Tracing indicator for the session. The buffer address field of the item can be specified as zero or any nonzero value. A value of zero indicates that trace is disabled. Any nonzero value indicates that trace is enabled.
FORMS$K_TRACEFILE	File specification of the file to be used to trace the execution of the session. The buffer address field of the item contains the address of a string that contains the trace file specification. If another request on this session opens the trace file, this item code is ignored.

send-record-message

OpenVMS usage:	char_string
type:	character-coded text string
access:	read only
mechanism:	by descriptor

Optional argument pointing to application data that is to be passed to the form. The form record is the description of this data to the Form Manager.

send-shadow-record

OpenVMS usage:	char_string
type:	character-coded text string
access:	read only
mechanism:	by descriptor

Optional argument that is an OpenVMS string descriptor that contains an indicator specifying whether to update last known values of form data items.

The send shadow record argument consists of a single 1-byte character. This character controls whether the Form Manager uses the data in the send record to update the last known values of the corresponding tracked form data items in the form.

If the character is N (either uppercase or lowercase), the Form Manager does not do the update; if the character is anything else, or if no send shadow record appears, the Form Manager does the update.

The length of *send-shadow-record* must be 1 byte.

If you do not pass *send-shadow-record*, the last known values of tracked form data items are updated.

For more information on shadow records, see Chapter 4.

Description

The SEND request causes the Form Manager to pass data from the application program record to form data items. The Form Manager can then display the form data items on the display device. Optionally, the Form Manager executes the SEND RESPONSE for the record or record list.

Return Values

FORMS$_BADARG

Bad argument or incorrect argument format for request.

FORMS$_BADRECCNT

The number specified in *record-count* in the request call does not match the number of records in the record list.

FORMS$_BADRECLEN

The length of the program record and the form record do not match. Either the program record descriptor length or the form record length is incorrect.

FORMS$_BAD_ SSHDWRECLEN

The request specified a length for *send-shadow-record* other than 1.

FORMS$_CANCELLED

FORMS$SEND processing interrupted by arrival of CANCEL request.

FORMS$_INVRECDES

One of the record message descriptors passed in FORMS$SEND is invalid.

FORMS$_NORECORD

FORMS$SEND specified a record identifier that is not in this form.

FORMS$_NORMAL

FORMS$SEND was completed.

FORMS$_NOSESSION

The session-identification string you passed in *session-id* does not match an existing session.

FORMS$_PENDING

An asynchronous operation you started has not yet completed.

FORMS$_TIMEOUT

Input was not completed in the specified time.

Examples

1.
```
CHARACTER*16 session_id
INTEGER forms_status,

    .
    .
    .
forms_status = forms$send (session_id, ! session id
1                     'account',   ! form record
2                     count,       ! number of records sent
3                     ,,           ! receive ctl text msg/ct
4                     ,,           ! send ctl text msg/ct
5                     ,            ! timeout
6                     ,            ! parent request id
7                     ,            ! request options item list
8                     descriptor1, ! info sent to form
9                     )            ! shadow rec
CALL check_forms_status( forms_status )
```

   The external request call in the preceding example causes the Form Manager to perform the following tasks:

   a. Validate the arguments in the request.

b. Verify that *session_id* names a valid session.

c. Copy the values stored in application program record fields to the form data items that correspond to the 'account ' form record.

d. Perform the SEND response, if defined.

e. Return control to the application program.

2.
```
CHARACTER*25 rec_ctrl_txt, send_ctrl_txt

INTEGER status, reclength, reccount, sendcount, timeout, one

    .
    .
    .
forms_status = forms$send  ( session_id,            ! session id
1                      'one_record',          ! form record
2                      one,                   ! number of records sent
3                      reccount,sendcount     ! receive ctl text msg/ct
4                      send_ctrl_txt,         ! send ctl text msg/ct
5                      rec_ctrl_txt,          ! rec ctl text msg/ct
6                      timeout,               ! timeout
7                      ,                      ! parent request id
8                      ,                      ! request options item list
9                      descriptor1,           ! info sent to form
10                     )                      ! shadow rec
CALL check_forms_status( forms_status )
```

The external request call in the preceding example causes the Form Manager to perform the following tasks:

a. Validate the arguments in the request.

b. Verify that *session_id* names a valid session.

c. Copy the values stored in application program record fields to the form data items that correspond to the 'one_record ' form record. The address of the application program record from which data is moved is specified by *descriptor1*.

d. Perform the control text responses named in *send_ctrl_txt*.

e. Perform the SEND response, if defined.

f. Store any receive control text items in *rec_ctrl_txt*.

g. Return control to the application program.

RECEIVE

RECEIVE — Copies data from form data items to an application program record.

Format

```
ret-status=FORMS$RECEIVE
```

```
(
session-id,
receive-record-name,
receive-record-count,
[receive-control-text, receive-control-text-count],
[send-control-text, send-control-text-count],
[timeout],
[parent-request-id],
[request-options],
[
[receive-record-message],
[receive-shadow-record],
] . . .
)
```

Returns

OpenVMS usage:	cond_value
type:	longword (unsigned)
access:	write only
mechanism:	by value

Arguments

session-id

OpenVMS usage:	char_string
type:	character-coded text string
access:	read only
mechanism:	by descriptor

Required argument that contains a unique session-identification string. The Form Manager returns the session-identification string to your application program during the processing of an ENABLE request.

You must pass *session-id* because the session-identification string identifies which DECforms session and form you want the Form Manager to use during this request.

The string you specify in this argument must be 16 bytes long.

receive-record-name

OpenVMS usage:	char_string
type:	character-coded text string
access:	read only
mechanism:	by descriptor

Required argument that is a single-byte character-string name of a form record (for single record transfers) or a form record list (for multiple record transfers). *Receive-record-name* is used to search for a form record or record list to use during data collection and trigger a RECEIVE RESPONSE within the form file.

receive-record-count

OpenVMS usage:	longword_unsigned
type:	longword (unsigned)
access:	read only
mechanism:	by reference

Required argument that contains the number of receive record items for this call. (A receive record item is made up of two parts: *receive-record-message* and *receive-shadow-record*. You cannot specify *receive-shadow-record* unless you also specify *receive-record-message*.)

Receive-record-count must match the number of records specified by *receive-record-name*. This value must be greater than 0.

receive-control-text

OpenVMS usage:	char_string
type:	character-coded text string
access:	write only
mechanism:	by descriptor

Optional argument that returns status information from the Form Manager to the application program when the request is terminated. The status information is divided into a maximum of five receive control text items.

Each receive control text item is five characters long, so the variable you pass in this argument must be able to store a string that is a multiple of five characters long. If you pass this argument, you must also pass *receive-control-text-count*.

If the length specified in the OpenVMS descriptor for *receive-control-text* is too short to hold all the receive control text items that the Form Manager attempts to return, the Form Manager discards those receive control text items that do not fit.

receive-control-text-count

OpenVMS usage:	longword_unsigned
type:	longword (unsigned)
access:	write only
mechanism:	by reference

Optional argument that specifies the number of receive control text items (each of which is five characters long) in the receive control message. This number indicates the number of receive control text items returned by the Form Manager to the application.

When the Form Manager terminates the request, it stores the number of receive control text items that it is returning in this argument. If you pass *receive-control-text*, you must pass this argument. This argument is set to 0 at the start of each request.

send-control-text

OpenVMS usage:	char_string
type:	character-coded text string
access:	read only
mechanism:	by descriptor

Optional argument in which you can pass zero to five send control text items to the Form Manager from the application program. Each send control text item must be one to five single-byte characters long and should name a control text response stored in the form. If you pass this argument, you must also pass the *send-control-text-count* argument.

You can pass a send control text item that is less than five characters long. However, if you are specifying an additional send control text item, the Form Manager requires you to pad the *send-control-text* with blanks to five 1-byte characters.

If you omit this argument, the Form Manager will not execute any control text response. If a send control text item specifies a control text response that does not exist, the Form Manager either processes the next send control text item or proceeds to the next request phase.

send-control-text-count

OpenVMS usage:	longword_unsigned
type:	longword (unsigned)
access:	read only
mechanism:	by reference

Optional argument in which you specify the number of control text items in *send-control-text*. You can specify zero to five control text items in this argument. If you pass *send-control-text*, you must pass this argument.

timeout

OpenVMS usage:	word_unsigned
type:	word (unsigned)
access:	read only
mechanism:	by reference

Optional argument in which you specify a timeout value for the entry timer. The entry timer controls the maximum number of seconds that can elapse between each operator keystroke.

The timeout value you pass in this argument supersedes any timeout value declared in the form.

If you omit this argument, the operator has unlimited time between each keystroke, unless you specify a timeout value in the form. If the timeout value is set to 0, the operator has unlimited time between each keystroke.

parent-request-id

OpenVMS usage:	char_string
type:	character-coded text string

access:	read only
mechanism:	by descriptor

Optional argument specifying the parent request of the new request. When a request that is currently executing invokes an escape routine that invokes another DECforms request, *parent-request-id* specifies the original request that called the escape routine.

This argument is intended only for when you issue DECforms calls from within an escape routine. There can be no *parent-request-id* argument for a request called from a nonprocedural escape routine called from within the application.

You can make *parent-request-id* available within an escape routine by passing the built-in form data item PARENTREQUESTID as one of the arguments to the escape routine within the CALL response step.

This argument is 24 bytes long.

Note

When an escape routine calls the current session or another session, the calling request hangs unless:

1. A built-in form data item of PARENTREQUESTID is declared as a form data item declaration.
2. The declared form data item is passed as a descriptor to the escape routine.
3. The escape routine passes it as a parameter to the request.

Once the built-in form data item declaration is declared and the PARENTREQUESTID is passed to the escape routine, the DECforms request being called from the escape routine is processed while the parent request completes.

request-options

OpenVMS usage:	item_list_2
type:	longword_unsigned
access:	read only
mechanism:	by reference

Optional item list containing request-specific arguments that control the environment of the request.

Request-options is the address of an OpenVMS item list containing zero or more request arguments. Each item consists of three longwords.

The first longword consists of two fields: a 1-word field that contains the length of the application buffer, and a 1-word field containing the item code value.

The second longword contains the address of the application buffer that either contains or will receive the data.

The third longword contains the address of a word to receive the length of the data written into the buffer. (The third longword is optional and is useful when the specified item code indicates that data will be returned to the application.)

The item list is terminated by a longword containing a 0.

Certain item codes correspond to logical names that the Form Manager translates during request processing. If you specify an item code, the Form Manager reads the item code directly and does not attempt to translate the corresponding logical names.

Table 5.4 contains the item codes that can be specified and their explanations.

Table 5.4. FORMS$RECEIVE Request Options

Item Code	Description
FORMS$K_ASTADR	The asynchronous system trap (AST) service routine to be executed when the request is complete. The buffer address field of the item contains the address of the entry mask for this routine.
FORMS$K_ASTPRM	The AST argument to be passed to the AST service routine. The buffer address field of the item contains the address of a longword value containing the value to be passed.
FORMS$K_EFN	Event flag the Form Manager sets when it completes the request. The buffer address field of the item contains the event flag number. The Form Manager clears the specified event flag at the beginning of the request.
FORMS$K_PRINTFILE	File specification of the file to be printed using the PRINT response step. The buffer address field of the item contains the address of a string that contains the print file specification.
FORMS$K_RSB	The quadword request status block to receive the completion status of the request. The buffer address field of the item contains the address of a DECforms quadword status block where the Form Manager writes the return status of the request. The Form Manager returns the final condition value of the request in the first longword of the status block. The second longword is reserved for future use.
FORMS$K_TRACE	Tracing indicator for the session. The buffer address field of the item can be specified as zero or any nonzero value. A value of zero indicates that trace is disabled. Any nonzero value indicates that trace is enabled.
FORMS$K_TRACEFILE	File specification of the file to be used to trace the execution of the session. The buffer address field of the item contains the address of a string that contains the trace file specification. If another request on this session opens the trace file, this item code is ignored.

receive-record-message

OpenVMS usage:	char_string
type:	character-coded text string
access:	write only
mechanism:	by descriptor

Optional argument pointing to form data that is to be passed to the application. The form record is the description of this data to the Form Manager. The Form Manager copies the values of corresponding form data items into this area of memory at the completion of the RECEIVE request.

If an exception condition occurs during processing of this request, the Form Manager does not move data to the application program.

receive-shadow-record

OpenVMS usage:	char_string
type:	character-coded text string
access:	write only
mechanism:	by descriptor

Optional argument that is a descriptor where the Form Manager writes modified field information that is returned to the application program from the form.

Receive-shadow-record contains information about each record field in the associated receive record message. The information specifies whether values in the record fields in the associated receive record message have changed from the values last known to the program. *Receive-shadow-record* contains one character for each record field in the receive record message, plus one additional character.

The length of *receive-shadow-record* is 1 plus the number of fields contained in the shadow record's associated form record. *Receive-shadow-record* contains a single character for every field in the record being returned.

The correspondence between the shadow record and the received record is by ordinal position of characters in the shadow record, with an offset of 1 because of the additional character at the start of the shadow record.

When the application program passes the RECEIVE request to the form, the first byte of *receive-shadow-record* can contain a character that specifies the following:

- N—The last known values are not updated.
- Any other character—The last known values are updated.

A character returned from the form to the program in the first byte of *receive-shadow-record* can be one of the following:

- 0—No fields in the record have been modified.
- X—Modified status was not requested for one field; all fields were either not tracked or not modified.

- 1—At least one field in the record has been modified.

Each subsequent character *receive-shadow-record* corresponds to a field in *receive-record-message* in the same relative position in the record.

The character is set as follows:

- 0—The corresponding field has not been modified.
- X—The field was not tracked by the Form Manager.
- 1—The corresponding field has been changed from its last known value.

Description

The RECEIVE request copies values from form data items to application program record fields. Optionally, the Form Manager executes the RECEIVE RESPONSE for the record or record list.

Return Values

FORMS$_BADARG

Bad argument or incorrect argument format for request.

FORMS$_BADRECCNT

The number specified in *record-count* in the request call does not match the number of records specified in the form.

FORMS$_BADRECLEN

The length of the program record and the form record do not match. Either the program record descriptor length or the form record length is incorrect.

FORMS$_BAD_RSHDWRECLEN

The request call specified a length for *receive-shadow-record* that does not match the length of *receive-shadow-record* in the form.

FORMS$_CANCELLED

FORMS$RECEIVE processing was interrupted by arrival of CANCEL request.

FORMS$_INVRECDES

Invalid record message descriptor. One of the record message descriptors passed in FORMS $RECEIVE is invalid.

FORMS$_NORECORD

FORMS$RECEIVE specified a record identifier that is not in this form.

FORMS$_NORMAL

FORMS$RECEIVE was completed successfully.

FORMS$_NOSESSION

The session-identification string you passed in *session-id* does not match an existing session.

FORMS$_PENDING

An asynchronous operation that you started has not yet been completed.

FORMS$_TIMEOUT

Input was not completed in the specified time.

Examples

1.
```
CHARACTER*16 session_id

    .
    .
    .
descriptor1.reclen  = 4 +
                      len(get_check.check_payto) +
                      len(get_check.check_memo)
descriptor1.address = %loc(get_check)

    .
    .
    .
forms_status = forms$receive( session_id, ! session id
1                             'get_check',! form record
2                             %ref(count),! # of records received
3                             ,,          ! receive ctl msg/ct
4                             ,,          ! send ctl msg/ct
5                             ,           ! timeout
6                             ,           ! parent request id
7                             ,           ! request options item
8                             descriptor1,! info received from form
9                             )           ! shadow record
                CALL check_forms_status( forms_status )
```

   The external request call in this language example causes the Form Manager to perform the following tasks:

   a. Validate the arguments in the request.

   b. Verify that *session-id* names a valid session.

   c. Perform the RECEIVE response.

   d. Copy the values stored in form data items that correspond to the ’get_check ’ record to the application program record. The address of the application program record is specified by *descriptor1*.

   e. Return control to the application program.

2.
```
CHARACTER*16 session_id

CHARACTER*25 rec_ctrl_txt, send_ctrl_txt
```

```
INTEGER status, reclength, reccount, sendcount, timeout

   .
   .
   .
forms_status = forms$receive (session_id,         ! session id
1                        'one_record',       ! form record
2                        one,                ! number of records
                                             ! received
3                     rec_ctrl_txt,reccount,! receive ctl text msg/ct
4                  send_ctrl_txt,sendcount, ! send ctl text msg/ct
5                        timeout,            ! timeout
6                        ,                   ! parent request id
7                        ,                   ! request options item list
8                        descriptor1,        ! info received from form
9                        )                   ! shadow rec
CALL check_forms_status( forms_status )
```

The external request call in the preceding example causes the Form Manager to perform the following tasks:

a. Validate the arguments in the request.

b. Verify that *session_id* names a valid session.

c. Perform the control text responses named in *send_ctrl_txt*.

d. Perform the RECEIVE RESPONSE.

e. Copy the values stored in form data items that correspond to the 'one_record' record to the application program record. The address of the application program record is specified by *descriptor1*.

f. Store any receive control text items in *rec_ctrl_txt*.

g. Return control to the application program.

TRANSCEIVE

TRANSCEIVE — Combines the functionality of the SEND and RECEIVE requests in a single request.

Format

```
ret-status=FORMS$TRANSCEIVE
           (
           session-id,
           send-record-name,
           send-record-count,
           receive-record-name,
           receive-record-count,
           [receive-control-text, receive-control-text-count],
           [send-control-text, send-control-text-count],
           [timeout],
           [parent-request-id],
           [request-options],
```

```
[
    [send-record-message],
    [send-shadow-record],
] . . .
[
    [receive-record-message],
    [receive-shadow-record],
] . . .
)
```

Returns

OpenVMS usage:	cond_value
type:	longword (unsigned)
access:	write only
mechanism:	by value

Arguments

See the sections on the SEND and RECEIVE requests for descriptions of the TRANSCEIVE arguments.

Description

The TRANSCEIVE request causes the Form Manager to combine the functionality of a SEND request and a RECEIVE request. First, the Form Manager moves data from application program record fields to form data items. When the request completes, the Form Manager then moves data from form data items to application program record fields.

Return Values

FORMS$_BADARG

Bad argument or incorrect argument format for request.

FORMS$_BADRECCNT

The number specified in the *record-count* argument in the request call does not match the number of records specified in the form.

FORMS$_BADRECLEN

The length of the program record and the form record do not match. Either the program record descriptor length or the form record length is incorrect.

FORMS$_BAD_RSHDWRECLEN

The request call specified a length for the *receive-shadow-record* argument that does not match the length of the *receive-shadow-record* argument in the form.

FORMS$_BAD_SSHDWRECLEN

The request specified a length for the *send-shadow-record* argument other than 1.

FORMS$_CANCELLED

FORMS$TRANSCEIVE processing was interrupted by arrival of CANCEL request.

FORMS$_INVRECDES

Invalid record message descriptor. One of the record message descriptors passed in FORMS $TRANSCEIVE is invalid.

FORMS$_NORECORD

FORMS$TRANSCEIVE specified a record identifier that is not in this form.

FORMS$_NORMAL

FORMS$TRANSCEIVE was completed successfully.

FORMS$_NOSESSION

The session-identification string you passed in *session-id* does not match an existing session.

FORMS$_PENDING

An asynchronous operation that you started has not yet been completed.

FORMS$_TIMEOUT

Input was not completed in the specified time.

Example

```
CHARACTER*16 session_id

    .
    .
    .
forms_status = forms$transceive(session_id,        ! session id
1                               'update',          ! send form record
2                               %ref(count),       ! number of records sent
3                               'operator_choice', ! receive form record
4                               %ref(count), ! number of records received
5                               ,,                 ! receive ctl txt/count
6                               ,,                 ! send ctl txt/count
7                               ,                  ! timeout
8                               ,                  ! parent request id
9                               ,                  ! request options
1                            descriptor1, ! program record data sent to form
2                            ,                  ! send shadow record
3                            descriptor2, ! program record data from form
4                            )                  ! recv shadow record
                CALL check_forms_status( forms_status )
```

The request call in this language example causes the Form Manager to perform the following tasks:

a. Validate the arguments in the request.

b. Verify that *session-id* names a valid session.

c. Copy the values stored in application program record fields to the form data items that correspond to the 'update ' form record. *Descriptor1* specifies the address of the application program record from which data is copied.

d. Perform the TRANSCEIVE response, if defined. If no TRANSCEIVE response is defined, perform the RECEIVE response for the receive record message, 'operator_choice'.

e. Copy the values stored in form data items that correspond to the 'operator_choice' form record to the application program record. *Descriptor2* specifies the address of the application program record.

f. Return control to the application program.

CANCEL

CANCEL — Cancels the current request and all outstanding requests for the specified session.

Format

```
ret-status = FORMS$CANCEL
          (session-id [,request-options] )
```

Returns

OpenVMS usage:	cond_value
type:	longword (unsigned)
access:	write only
mechanism:	by value

Arguments

session-id

OpenVMS usage:	char_string
type:	character-coded text string
access:	read only
mechanism:	by descriptor

Required argument that contains a unique session-identification string. The Form Manager returns the session-identification string to your application program during the processing of an ENABLE request.

You must pass *session-id* because the session-identification string identifies which session you want the Form Manager to cancel.

The string you specify must be 16 bytes long.

request-options

OpenVMS usage:	item_list_2
type:	longword (unsigned)

access: read only

mechanism: by reference

Optional item list containing request-specific arguments that control the environment of the request.

Request-options is the address of an OpenVMS item list containing zero or more request arguments. Each item consists of three longwords.

The first longword consists of two fields: a 1-word field that contains the length of the application buffer, and a 1-word field containing the item code value.

The second longword contains the address of the application buffer that either contains or will receive the data.

The third longword contains the address of a word to receive the length of the data written into the buffer. (The third longword is optional and is useful when the specified item code indicates that data will be returned to the application.)

The item list is terminated by a longword containing a 0.

Certain item codes correspond to logical names that the Form Manager translates during request processing. If you specify an item code, the Form Manager reads the item code directly and does not attempt to translate the corresponding logical names.

Table 5.5 contains the item codes that can be specified and their explanations.

Table 5.5. FORMS$CANCEL Request Options

Item Code	Description
FORMS$K_ASTADR	The asynchronous system trap (AST) service routine to be executed when the request is complete. The buffer address field of the item contains the address of the entry mask for this routine.
FORMS$K_ASTPRM	The AST argument to be passed to the AST service routine. The buffer address field of the item contains the address of a longword value containing the value to be passed.
FORMS$K_EFN	Event flag the Form Manager sets when it completes the request. The buffer address field of the item contains the event flag number. The Form Manager clears the specified event flag at the beginning of the request.
FORMS$K_RSB	The quadword request status block to receive the completion status of the request. The buffer address field of the item contains the address of a DECforms quadword status block where the Form Manager writes the return status of the request. The Form Manager returns the final condition value of the request in the first longword of the status block. The second longword is reserved for future use.

Description

The CANCEL request causes the Form Manager to cancel the processing of any request that is in progress for the specified session and any outstanding request for that session. The cancelled requests end with the status FORMS$_CANCELLED.

CANCEL executes asynchronously with respect to the requests that it cancels; CANCEL returns to the routine that called it before it cancels the specified request.

Calling FORMS$CANCEL directs the Form Manager to terminate the current request as quickly as possible. No additional input or output to the screen is started nor is any data returned to the program.

Because a cancelled request may have been outputting data to the screen at the time it was cancelled, the contents of the display may not accurately reflect the current values of form data as stored in the form.

To ensure that the screen is correct after you cancel a request, issue the REFRESH response step or press Ctrl/W to refresh the screen.

FORMS$CANCEL should be restricted to cases where the operator has clear and unmistakable feedback from the form that a request has been cancelled.

Return Values

FORMS$_BADARG

Bad argument or incorrect argument format for request.

FORMS$_CANINPROG

A previous CANCEL request is in progress.

FORMS$_NOACTREQ

No active request to cancel.

FORMS$_NORMAL

FORMS$CANCEL was completed successfully.

FORMS$_NOSESSION

No session was active.

Example

```
CHARACTER*16 session_id
INTEGER forms_status

    .
    .
    .
forms_status = FORMS$CANCEL(session_id,)
```

The request call in the preceding example would cause the Form Manager to perform the following tasks:

a. Validate the arguments in the request.

b. Verify that *session_id* names a valid session.

c. Cancel the request.

d. Return control to the application program.

Chapter 6. Using the Portable API

This chapter describes the DECforms application programming interface (API) for C and FORTRAN.

The portable API interface supports the following languages:

- VAX C
- VAX FORTRAN

When you convert large applications from the OpenVMS API to either the C or FORTRAN API syntax, you may need to use portions of both interfaces temporarily. However, for applications running in a production environment, you should avoid mixing the two interfaces because signal handling, parameter validations, and other functions are designed differently to solve different problems.

The portable API bindings are designed to provide a simple DECforms interface.

Table 6.1 contains a list of the available routine parameters.

Table 6.1. Portable API Routine Parameters

Routine Parameter	Request
Form Name	ENABLE
Form File Name	ENABLE
Device Name	ENABLE
Session Id	All requests
Send Record Name	SEND and TRANSCEIVE
Send Record Structure	SEND and TRANSCEIVE
Receive Record Name	RECEIVE and TRANSCEIVE
Receive Record Structure	RECEIVE and TRANSCEIVE
Request Option Structure	All requests

The definitions for the Form Name, Form File Name, Device Name, Session Id, and Send and Receive Record Name parameters remain unchanged from the DECforms OpenVMS API; however, the data types are different.

In the portable API, you pass records by using a predefined structure named Forms_Record_Data. You specify optional parameters by using a predefined structure named Forms_Request_Options.

A DECforms portable library file defining all the literals and type definitions is provided for DECforms portable application programs that use the C and FORTRAN languages:

- formsdef.h—for C programs
- formsdef.f—for FORTRAN programs

These files contain external routine declarations for the bindings, type definitions for C binding users, literal definitions, and structure definitions for Forms_Record_Data and Forms_Request_Options.

A copy of these files exists in SYS$LIBRARY.

6.1. Using the Forms_Record_Data Structure

Forms_Record_Data is a structure containing four elements. These elements are listed in Table 6.2.

Table 6.2. Forms_Record_Data Elements

Element	Required or Optional
Data_record	Required
Data_length	Required
Shadow_record	Optional
Shadow_length	Optional

Forms_Record_Data stores either send or receive data record information and shadow record information. This structure is a required parameter for the RECEIVE and TRANSCEIVE request calls.

To pass records, record parameters such as send record and receive record in a request call are used to pass the address of a single structure or an array structure of Forms_Record_Data. To pass a single record, you do not need to specify the record count to DECforms. DECforms sets the record count to 1 by default. If you pass zero or more than one record, you must specify the record count representing the actual number of records being passed in Forms_Request_Options.

For example, to pass two records, declare and use the array as follows:

C Binding

```
#include
<formsdef.h>

void sample_send( Forms_Session_Id session_id )

{
     Forms_Request_Options request_options[2];
     Forms_Record_Data     send_records[2];
     Forms_Status          forms_status;
     struct_record {
           int a;
           int b;
           } record1 = {1,2};

     struct_record2 {
           int a;
           int b;
           int c;
           } record2 = {3,4,5};

     char shadow2[1];

     /*                                          */
     /* Setting up send records                  */
     /*                                          */

     send_records[0].data_record    = &record1;
     send_records[0].data_length    = sizeof (record1);
     send_records[0].shadow_record  = NULL;
     send_records[0].shadow_length  = 0;
```

```
    send_records[1].data_record    = &record2;
    send_records[1].data_length    = sizeof(record2);
    send_records[1].shadow_record  = &shadow2;
    send_records[1].shadow_length  = 4;

    /*                                          */
    /* Specifying record count in request option */
    /*                                          */

    request_options[0].option = forms_c_opt_send_record;
    request_options[0].send_record.count= 2;

    request_options[1].option = forms_c_opt_end;

    forms_status = forms_send(  session_id,
                                record_list_name,
                                send_records,
                                request_options);
    check_forms_status (forms_status)
}
```

FORTRAN Binding

```
            SUBROUTINE sample_send (session_id)

            INCLUDE 'formsdef.f'

        C
        C  Declare record1 and record2
        C
               Character*16 session_id
        RECORD /Forms_Record_Data/ send_records(2)
        RECORD /Forms_Request_Options/ request_options(2)

               STRUCTURE  /r1/
                  INTEGER a
                  INTEGER b
               END STRUCTURE

               STRUCTURE  /r2/
                  INTEGER a
                  INTEGER b
                  INTEGER c
               END STRUCTURE

               RECORD /r1/ record1
               RECORD /r2/ record2
               CHARACTER*1 shadow2
               INTEGER forms_status

         C
         C Init record1 and record2
         C

               record1.a = 1
               record1.b = 2
               record2.a = 3
```

```
      record2.b = 4
      record2.c = 5

C
C Set up send record
C

      send_records(1).data_length   = 8
      send_records(1).data_record   = %LOC (record1)
      send_records(1).shadow_length = 0
      send_records(1).shadow_record = 0

      send_records(2).data_length   = 12
      send_records(2).data_record   = %LOC (record2)
      send_records(2).shadow_length = 4
      send_records(2).shadow_record = %LOC (shadow2)

C
C Set up record count
C
      request_options(1).option = forms_c_opt_send_record
      request_options(1).send_record_count = 2

      request_options(2).option = forms_c_opt_end

      forms_status = forms_send_for (
   +                     session_id,
   +                     record_list_name,
   +                     send_records,
   +                     request_options)

      CALL check_form_status (forms_status)
      END
```

6.2. Using the Forms_Request_Options Structure

Forms_Request_Options is a union that contains groups of small structures and a special element named *option*. To set up a request option list, you declare an array of at least two elements of type Forms_Request_Options and specify the name for the structure to be used in the option variable. To end a request option list, assign the symbolic constant, forms_c_opt_end, to the option variable in the last array element.

For example, to specify a print file name, declare the following:

C Binding

```
# define PRINT_FILE_NAME "MY_FORM.TXT"

Forms_Request_Options enable_request_options[2];

enable_request_options[0].option = forms_c_opt_print;
enable_request_options[0].print.file_name = PRINT_FILE_NAME;
enable_request_options[0].print.file_name_length =
strlen(PRINT_FILE_NAME);
```

```
enable_request_options[1].option = forms_c_opt_end;
```

FORTRAN Binding

```
CHARACTER*(*) print_file_name
PARAMETER     (print_file_name='SYS$SCRATCH:FORMS$CHECKING_FORM.TXT')

RECORD /Forms_Request_Options/ enable_request_options(2)

enable_request_options(1).option = forms_c_opt_print
enable_request_options(1).print_file_name = %LOC (print_file_name)
enable_request_options(1).print_file_name_length = LEN(print_file_name)

enable_request_options(2).option = forms_c_opt_end
```

Certain option codes correspond to logical names that the Form Manager translates during request processing. If you specify an option code, the Form Manager reads the option code directly and does not attempt to translate the corresponding logical names. Table 6.3 lists the request options that you can use in the portable API requests. The names in the leftmost column (Option Type Code) are used in the format forms_c_opt_ *code* to describe the name of the request option and in the format . *code* in the option declaration. The names in the next column (Field) are elements for each option structure. (To see how the options are declared, see the examples in this section.)

Table 6.3. Request Options for the Portable API

Option Type Code	Field	Request Type	Description
send_record	count	SEND TRANSCEIVE	Number of data records to be passed from the application program to the form
send_control	text	SEND TRANSCEIVE	Control text to be passed to the form
	text_count	SEND TRANSCEIVE	Number of control text items to be passed to the form
receive_record	count	RECEIVE TRANSCEIVE	Number of data records to be passed from the form to the application program
receive_control	text	RECEIVE TRANSCEIVE	Control text to be returned by the form
	text_count	RECEIVE TRANSCEIVE	Number of control text items to be returned by the form
trace	file_name	all	Name of trace file that receives output trace logging
	file_name_length	all	Length of trace file name (number of characters)

Option Type Code	Field	Request Type	Description
	flag	all	Flag to turn trace on or off
print	file_name	all	Name of print file that receives output text
	file_name_length	all	Length of print file name (number of characters)
language	name	ENABLE	Name of language to match during layout selection
	name_length	ENABLE	Length of language name (number of characters)
timeout	period	all	Describes timeout period; limits to 32767 seconds or less
completion_routine	address	all	Address of completion routine—equivalent to FORMS$K_ASTADR and FORMS $K_ASTPRM in the OpenVMS API
	parameter	all	Completion routine parameter
completion_status	address	all	Address of completion status—equivalent to FORMS$K_RSB in the OpenVMS API
parent_request	id	all	Session id of parent request—used in call to escape routine
form	object	ENABLE	Form object for linked form and table
no_term_io	flag	DISABLE SEND RECEIVE TRANSCEIVE	Suppresses terminal I/O
selection_label	name	ENABLE	Label name for a PRINTER layout
	name_length	ENABLE	Length of label name string (number of characters)

You must pass all character string elements in Forms_Request_Options by reference. Each string element, except for *parent_request_id*, *send_control_text* and *receive_control_text*, has a corresponding string length element. You must specify the length of each string element either by using the string length element corresponding to it or by terminating the string with a null. If you terminate a string with a null and specify the corresponding string length to a nonzero value, DECforms uses the corresponding string length.

Parent_request_id and *receive_control_text* are pointers to fixed length elements in Forms_Request_Options. *Parent_request_id* should be a pointer to a string of 24 characters, and *receive_control_text* must be a pointer to a string of 25 characters. DECforms assumes that you have allocated enough memory to store information in these elements.

The receive control text count in Forms_Request_Options indicates how many control text items are written into the receive control text string. Each control text item is five characters in length. Use *send_control_text_count* to specify how many control text items are passed to DECforms.

In Forms_Request_Options, *timeout* is defined as type integer. However, due to a restriction in the OpenVMS terminal driver, the *timeout* value for any request is limited to the range between 0 and 32,767 seconds.

6.3. Using Disk-Based Forms or Linked Forms

With the DECforms portable API bindings, your program must enable a separate disk-based binary form. With the OpenVMS API, you can either enable a disk-based form or link the form in with your program.

The portable API binding does not support the use of shareable forms because the use of shareable images is not a portable concept.

To enable a disk-based form, specify your binary form name in the ENABLE request call in the program. (If you leave out the file type, the Form Manager looks for .form.)

For example:

C Binding

```
status = forms_enable  (session_id,          /* session ID */
                        NULL,                /* current device */
                        "my_form_file",      /* name of form file */
                        ,                    /* no form name specified*/
                        );                   /* no request options */
```

FORTRAN Binding

```
 status = forms_enable_for (session_id,,
+                           device_name,
+                           'my_form_file',
+                           ,
+                           );
```

To enable a linked form, do the following:

1. Use the DECforms Extract Object Utility to create a form object file from your binary form file.

   Specify the /PORTABLE_API qualifier in the FORMS EXTRACT OBJECT command to convert form files into form object files for the DECforms API bindings.

2. Declare an external global variable using the name of your form.

Do this by using the name of the form with the type Forms_Form_Object in the C binding or EXTERNAL INTEGER in the FORTRAN binding.

The Linker resolves the value of this global variable by using information produced by the DECforms Extract Object Utility.

3. Set up in Forms_Request_Options a form object option that uses the global variable.

4. Specify your form name and the request option in the ENABLE request call in the program.

For example:

C Binding

```
Forms_Form_Object      my_form_name;
Forms_Session_Id       session_id;
Forms_Request_Options  enable_options[2];

enable_options[0].option = forms_c_opt_form;
enable_options[0].form.object = my_form_name;
enable_options[1].option = forms_c_opt_end;

status = forms_enable  (session_id,,
                        device_name,
                        ,
                        "my_form_name",
                        enable_options);
```

FORTRAN Binding

```
INTEGER         my_form_name
EXTERNAL        my_form_name

CHARACTER*(16)  session_id
RECORD /Forms_Request_Options/ enable_options(2)

enable_options(1).option = forms_c_opt_form
enable_options(1).form_object = %LOC (my_form_name)
enable_options(2).option = forms_c_opt_end;

 status = forms_enable_for (session_id,
+                           device_name,
+                           ,
+                           'my_form_name',
+                           enable_options);
```

6.4. Using Escape Routines

You must link escape routines with the application program when you use the DECforms portable API. To resolve escape routine references in your form, you must create a form object file by using the Forms Extract Object Utility. The form object file resolves the escape routine references when escape routines are linked in. You then link the form object file with your escape routines and application program.

If you plan to use a disk-based form, you should use the /NOFORM_LOAD qualifier with the FORMS EXTRACT OBJECT command. If you plan to use a linked form, use the /FORM_LOAD qualifier when you create your form object.

For more information about escape routines, see Section 2.3. For an example of linking escape routines with your application program, see the *VSI DECforms Guide to Developing an Application*. For a description of the FORMS EXTRACT OBJECT command, see the *VVSI DECforms Guide to Commands and Utilities*.

6.5. Using Error Message Routines

DECforms provides an error message routine named forms_errormsg() for C and forms_errormsg_for() for FORTRAN. These routines translate return values from a DECforms request into meaningful message text. These routines require a request status number and a string pointer pointing to pre-allocated memory large enough to store up to 256 characters. DECforms stores the message text in the location specified by the given string pointer.

The following examples show routine descriptions for both C and FORTRAN bindings.

Routine Description for C

```
char msg_text_ptr[256];
void forms_errormsg ( Forms_Status   errno,
                      Forms_Text_Ptr msg_text_ptr);
```

Routine Description for FORTRAN

```
SUBROUTINE forms_errormsg_for ( INTEGER           errno,
                                CHARACTER*(256)   msg_text_ptr)
```

6.6. Referencing Error Numbers

There are three types of errors that can occur during a call to the portable C or FORTRAN interface:

- FIMS-defined errors
- DECforms defined errors
- User-defined errors

User-defined errors are passed using the control text fields in the Forms_Request_Options union structure. Errors related to the FIMS standard and to DECforms software are passed as status values returned by the request calls. The status values are also called FIMS error numbers. Each number is a unique integer associated with some request condition.

An error number is a positive or negative decimal number. Positive numbers represent a status defined by FIMS. Negative numbers represent a status specific to DECforms.

Successfully processed requests receive a status value of 0. Literals are defined in the formsdef.h and formsdef.f header file for referring to these status numbers; for example, forms_s_normal = 0. You can find the meaning of each status number in the header files.

Table 6.4 lists the error numbers, definitions, corresponding control texts, and matching symbolic constants. Table 6.4 should only be used as reference. For the most up-to-date list, refer to the formsdef.h or formsdef.f header file that ships in your kit.

Table 6.4. Error Numbers

Definition	Error Number	Control Text	Description
forms_s_normal	0	S000	Request calls completed successfully.

Definition	Error Number	Control Text	Description
forms_s_timeout	1	ES001	Input was not completed in the specified time.
forms_s_formerror	2	ES002	Encountered problem when using form file.
forms_s_nolayout	3	ES003	No layout fit terminal type, language, and display size.
forms_s_invdevice	4	ES004	Invalid device specified in ENABLE.
forms_s_hangup	5	ES005	Data set hangup; session disabled.
forms_s_norecord	7	ES007	Specified record identifier not in form.
forms_s_badreclen	8	ES008	Record length argument does not match length of record in form.
forms_s_inuse	10	ES010	Attempted to disable a form still in use.
forms_s_nosession	11	ES011	*session_id* does not match existing session.
forms_s_return_immed	12	ES012	Request terminated due to REQUEST IMMEDIATE.
forms_s_nodecpt	14	ES014	Decimal or comma decimal point positioned incorrectly in record field.
forms_s_bad_rshdwreclen	15	ES015	Length of *receive-shadow-record* does not match length specified in form.
forms_s_bad_sshdwreclen	16	ES016	Length of *send-shadow-record* is something other than 1.
forms_s_cancelled	17	ES017	Request interrupted by arrival of CANCEL.
forms_s_noactreq	19	ES019	No active requests to CANCEL.
forms_s_invlobound	24	ES024	Subscript reference less than base.
forms_s_invhibound	25	ES025	Subscript reference greater than array dimension defined.
forms_s_illdtcvt	–1	EI001	Illegal DATE, TIME, ADT conversion.

Definition	Error Number	Control Text	Description
forms_s_badreccnt	–2	EI002	Number of records does not match number specified in record list.
forms_s_converr	–3	EI003	Error while converting from one data type to another.
forms_s_aborted	–4	EI004	Session ended abnormally due to severe error in another request.
forms_s_badarg	–5	EI005	Bad argument or incorrect format.
forms_s_baditmlstcode	–6	EI006	Invalid item code found in item list.
forms_s_baditmlstcomb	–7	EI007	Invalid combination of item codes found in item list.
forms_s_blocked_by_ast	–8	EI008	Cannot process request; block by application AST.
forms_s_bad_devhlr	–9	EI009	Invalid device handler.
forms_s_caninprog	–10	EI010	Previous CANCEL is in progress.
forms_s_cantopendic	–46	EI046	Cannot open dictionary.
forms_s_closetrace	–11	EI011	Cannot close trace file.
forms_s_deverr	–12	EI012	Device I/O error.
forms_s_disinprog	–13	EI013	Previous DISABLE is in progress.
forms_s_exprevalerr	–14	EI014	Cannot convert operands into common data type.
forms_s_fatinterr	–15	EI015	Fatal internal error.
forms_s_illctltxtcnt	–16	EI016	Illegal control text count argument.
forms_s_illfldvaluectx	–17	EI017	Illegal FIELDVALUE context.
forms_s_illvpuse	–18	EI018	Illegal use of print viewport.
forms_s_intdatcor	–19	EI019	Database consistency check failed.
forms_s_invrange	–20	EI020	Invalid subscript range.
forms_s_invreccnt	–21	EI021	Invalid record count value.
forms_s_invrecdes	–22	EI022	Invalid record message descriptor.

Definition	Error Number	Control Text	Description
forms_s_nohandler	–23	EI023	No device handler for such device.
forms_s_nolicense	–24	EI024	No DECforms software license is active.
forms_s_noparent	–25	EI025	Specified parent request does not exist.
forms_s_no_read_access	–26	EI026	No read access to user argument.
forms_s_openout	–27	EI027	Cannot open specified output file.
forms_s_opentrace	–28	EI028	Cannot open trace file for output.
forms_s_paramovrflow	–29	EI029	Escape routine parameter has overflowed.
forms_s_procesc_not_found	–30	EI030	Cannot find address of procedural escape.
forms_s_proc_escape_error	–31	EI031	Request ended due to severe error in escape routine.
forms_s_recvrecitems	–32	EI032	Number of receive record items does not match record count value.
forms_s_reqdarg	–33	EI033	Required argument missing.
forms_s_sendrecitems	–34	EI034	Number of send record items does not match record count value.
forms_s_strtooshort	–35	EI035	Length of string is too small.
forms_s_writetrace	–36	EI036	Cannot write to trace file.
forms_s_no_write_access	–37	EI037	No write access to user argument.
forms_s_bckgrndio	–38	EI038	Attempted read or write I/O from background process.
forms_s_timeract	–39	EI039	Attempted timed field input while alarm active.
forms_s_blkbyreq	–40	EI040	Attempted synchronous request while another request active.
forms_s_imgmismatch	–41	EI041	Shareable image mismatch.

Definition	Error Number	Control Text	Description
forms_s_nyi	–42	EI042	Requested operation is not yet implemented.

6.7. C and FORTRAN Request Calling Description

This section describes each DECforms C and FORTRAN request in a structured format that provides the following information:

- The name of the request and a sentence describing its purpose
- The syntax of the request
- A description of each argument
- A complete description of the request
- An example of using each request in the FORTRAN and C bindings

For examples of how to include requests in an application program, see Chapter 2. For information on how the Form Manager processes requests, see Chapter 4.

ENABLE

ENABLE — Initializes a DECforms session.

C Binding

```
#include <formsdef.h>
Forms_Status forms_enable (
                      Forms_Session_Id session_id,
                      Forms_Text_Ptr device_name,
                      Forms_Text_Ptr form_file_name,
                      Forms_Text_Ptr form_name,
                      Forms_Request_Options request_options[ ] );
```

FORTRAN Binding

```
INCLUDE 'FORMSDEF.F'
INTEGER*4 FUNCTION forms_enable_for ( session_id,
                                      device_name, form_file_name,
                                      form_name, request_options )

CHARACTER*16                     session_id
CHARACTER*(*)                    device_name
CHARACTER*(*)                    form_file_name
CHARACTER*(*)                    form_name
RECORD /Forms_Request_Options/   request_options
```

Arguments

session_id

type:	character-coded text string

access:	write only
mechanism:	by reference

Required argument that is filled by the Form Manager. In this argument, the Form Manager returns a 16-byte character session identification string. *Session_id* associates the display device with the form that was loaded into memory.

You must pass this session identification string in subsequent request calls to tell the Form Manager which display device and form you want the Form Manager to use.

device_name

type:	character-coded text string
access:	read only
mechanism:	by reference

Required argument that is a character string name for the device to be used as the display device during this session.

You can define a logical name for the device name and pass that logical name in *device_name*: you can specify SYS$INPUT or a file name as the device name to use a terminal or a DDIF output file, respectively.

When you do not specify *device_name* or you specify *device_name* as NULL in the forms_enable call, a FORMS$DEFAULT_DEVICE logical name is available for specifying *device_name*. If *device_name* is omitted or NULL and FORMS$DEFAULT_DEVICE is undefined, SYS$INPUT is used.

form_file_name

type:	character-coded text string
access:	read only
mechanism:	by reference

Specifies the character string file specification of the form file to be used during this session. This argument is optional. The file specification can include:

node name
device name
directory specification
file name
file type
file version number

If you omit the node name, device name, or directory specification, the Form Manager uses the node you are logged into, the current default device, and the current default directory, respectively.

Using the portable API:

- You can define a logical name for the file specification and pass that logical name in *form_file_name*.
- If you do not specify *form_file_name*, you must specify *form_name*.
- If you omit the file version number, the Form Manager uses the latest version (highest version number) of the specified file.

- If you omit the file type, the Form Manager looks for a file with the .form type.

form_name

type:	character-coded text string
access:	read only
mechanism:	by reference

Specifies the name of the form in the IFDL source file.

If you do not specify *form_name*, you must specify *form_file_name*.

request_options

type:	Forms_Request_Options
access:	read only
mechanism:	by reference

Optional item list containing request-specific arguments that control the environment of the request.

For more information about request options, see Section 6.2.

Table 6.5 contains the Request Option codes that can be specified and their explanations.

Table 6.5. FORMS_ENABLE Request Options

Item Code	Description
forms_c_opt_selection_label	Selection label value for PRINTER layout
forms_c_opt_default_color	Default color type
forms_c_opt_default_term	Default terminal type
forms_c_opt_no_term_io	Suppress terminal IO
forms_c_opt_form	DECforms Form Object
forms_c_opt_trace	Name of trace file and flag
forms_c_opt_print	Name of print file to use
forms_c_opt_language	Name of language to match
forms_c_opt_completion_status	Location of store completion status
forms_c_opt_completion_routine	Completion routine and parameter
forms_c_opt_end	End of Request Options
forms_c_opt_parent_request	Specifies *parent_request_id*
forms_c_opt_receive_control	Receive control text information
forms_c_opt_send_control	Send control text information
forms_c_opt_timeout	Timeout period

Description

The ENABLE request loads the specified form file into memory, selects an appropriate layout, and attaches the specified display device. The Form Manager also returns a unique session-identification string and initializes all form data items. For more information about initializing an ENABLE request, see Section 4.1.2.

You can include more than one ENABLE request call in your application program because the Form Manager supports multiple synchronous requests to display devices. For example, the operator's device can be enabled for two different sessions. The Form Manager can perform operations in any enabled session.

Examples

1. ```
   status = forms_enable(
                        session_id,     /* Session id returned */
                        NULL,           /* Current device      */
                        "myform",       /* Name of form file   */
                        NULL,           /* Name of the form    */
                        NULL);          /* No request Options  */
   ```

   This C binding example enables a form, provides the form file name, specifies a NULL device name, and uses no request options.

2. ```
   #include
   <formsdef.h>
   #if defined(MSDOS)
   #include
   <windows.h>
   #else
   #include
   <stdio.h>
   #endif

   #define PRINT_FILE_NAME "monthly_report"
   #define OTHER_PRINT_FILE "weekly_report"

   #if defined(MSDOS)
   int PASCAL WinMain(HANDLE hInstance, HANDLE hPrevInstance,
            LPSTR lpszCmdLine, int cCmdShow)
   #else
   main()
   #endif
   {
   Forms_Session_Id    session_id;
   Forms_Request_Options enable_options[3];
   Forms_Status status;

   /* To generate the monthly report: */

   enable_options[0].option = forms_c_opt_selection_label;
   enable_options[0].selection_label.name = PRINT_FILE_NAME;
   enable_options[0].selection_label.name_length = strlen(PRINT_FILE_NAME);

   enable_options[1].option = forms_c_opt_end;

   status = forms_enable(session_id,
   #if defined(MSDOS)
           "%PRINTER",
   #else
           "printed_form.doc",
   #endif
           "myform",
           NULL,
           enable_options);
   ```

```
if (status != forms_s_normal)
    {
    return;
    }

forms_disable(session_id, 0);

/* To generate the weekly report: */

enable_options[0].option = forms_c_opt_selection_label;
enable_options[0].selection_label.name = OTHER_PRINT_FILE;
enable_options[0].selection_label.name_length =
 strlen(OTHER_PRINT_FILE);

enable_options[1].option = forms_c_opt_end;

status = forms_enable(session_id,
#if defined(MSDOS)
        "%PRINTER",
#else
        "printed_form.doc",
#endif
        "myform",
        NULL,
        enable_options);
if (status != forms_s_normal)
    {
    return;
    }

forms_disable(session_id, 0);
}
```

Written in the C binding, this example:

- Enables a form.
- Provides the form file name.
- Specifies the %PRINTER device on the PC and the DDIF output file name printed_form.doc on OpenVMS.
- Specifies the name of the form file.
- Specifies a NULL form name.
- Uses request options to select between two PRINTER layouts with LAYOUT SELECTION clauses labeled "monthly_report" and "weekly_report" in the IFDL source file.

3.
```
#include
<formsdef.h>
#include
<stdio.h>

#define PRINT_FILE_NAME "monthly_report"
#define OTHER_PRINT_FILE "weekly_report"

main()
```

```
{
Forms_Session_Id    session_id;
Forms_Request_Options enable_options[3];
Forms_Status status;

/* To generate the monthly report: */

enable_options[0].option = forms_c_opt_selection_label;
enable_options[0].selection_label.name = PRINT_FILE_NAME;
enable_options[0].selection_label.name_length = strlen(PRINT_FILE_NAME);

enable_options[1].option = forms_c_opt_end;

status = forms_enable(session_id,

        "printed_form.doc",
        "myform",
        NULL,
        enable_options);
if (status != forms_s_normal)
    {
    return;
    }

forms_disable(session_id, 0);

/* To generate the weekly report: */

enable_options[0].option = forms_c_opt_selection_label;
enable_options[0].selection_label.name = OTHER_PRINT_FILE;
enable_options[0].selection_label.name_length =
 strlen(OTHER_PRINT_FILE);

enable_options[1].option = forms_c_opt_end;

status = forms_enable(session_id,
        "printed_form.doc",
        "myform",
        NULL,
        enable_options);
if (status != forms_s_normal)
    {
    return;
    }

forms_disable(session_id, 0);
}
```

Written in the C binding, this example:

- Enables a form.
- Provides the form file name.
- Specifies the DDIF output file name printed_form.doc.

- Specifies the name of the form file.
- Specifies a NULL form name.
- Uses request options to select between two PRINTER layouts with LAYOUT SELECTION clauses labeled "monthly_report" and "weekly_report" in the IFDL source file.

4.
```
print_file = 'forms_checking_form.txt'
request_options(1).option =  forms_c_opt_print
request_options(1).print_file_name = %LOC (print_file)
request_options(1).print_file_name_length = 23
request_options(2).option = forms_c_opt_end

status = forms_enable_for(
+                    session_id,      ! session id
+                      ,              ! current device
+                    'my_forms',      ! name of form file
+                      ,
+                    request_options)
```

This FORTRAN example shows how to set a request option in the print file name by using a file whose name is in a string variable.

DISABLE

DISABLE — Terminates a session.

C Binding

```
#include <formsdef.h>
Forms_Status forms_disable (
                      Forms_Session_Id session_id,
                      Forms_Request_Options request_options[ ] );
```

FORTRAN Binding

```
INCLUDE 'FORMSDEF.F'
INTEGER*4 FUNCTION forms_disable_for ( session_id, request_options )

CHARACTER*16                      session_id
RECORD /Forms_Request_Options/    request_options
```

Arguments

session_id

type:	character-coded text string
access:	read only
mechanism	by reference

Required argument that contains a unique session-identification string. The Form Manager returns the session-identification string to your application program during the processing of an ENABLE request.

You must pass this argument because the session-identification string identifies which display device and session you want the Form Manager to use during this request.

request_options

type:	Forms_Request_Options
access:	read only
mechanism	by reference

Optional item list containing request-specific arguments that control the environment of the request.

For more information about request options, see Section 6.2.

Table 6.6 shows the request options that you can specify.

Table 6.6. FORMS_DISABLE Request Options

Item Code	Description
forms_c_opt_trace	Name of trace file and options
forms_c_opt_completion_status	Location of store completion status
forms_c_opt_completion_routine	Completion routine and parameter
forms_c_opt_end	End of request options
forms_c_opt_receive_control	Receive control text information
forms_c_opt_send_control	Send control text information
forms_c_opt_timeout	Timeout period

Description

The DISABLE request terminates the specified session. The Form Manager detaches the display device and the form that are associated with the session-identification string. Once the Form Manager completes the processing of this request, the session specified by the session-identification string no longer exists. The values of all form data items in the form are disabled last.

Example

```
status = forms_disable ( session_id, NULL);
```

This C example disables a form and does not pass any request options.

SEND

SEND — Copies data from an application program record to form data items.

C Binding

```
#include <formsdef.h>
Forms_Status forms_send (
                        Forms_Session_Id        session_id,
                        Forms_Text_Ptr          send_record_name,
```

```
                        Forms_Record_Data       send_record[ ],
                        Forms_Request_Options   request_options[ ] );
```

FORTRAN Binding

```
INCLUDE 'FORMSDEF.F'

INTEGER*4 FUNCTION forms_send_for ( session_id, send_record_name,
                                    send_record, request_options )

CHARACTER*16                        session_id
CHARACTER*(*)                       send_record_name
RECORD /Forms_Record_Data/          send_record
RECORD /Forms_Request_Options/      request_options
```

Arguments

session_id

type:	character-coded text string
access:	read only
mechanism:	by reference

Required argument that contains a unique session-identification string. The Form Manager returns the session-identification string to your application program during the processing of an ENABLE request.

You must pass this argument because the session-identification string identifies which DECforms session and form you want the Form Manager to use during this request.

send_record_name

type:	character-coded text string
access:	read only
mechanism:	by reference

Required argument that is a single-byte character string name of the form record (for single record transfers) or record list (for multiple record transfers) to which the Form Manager is passing data. *Send_record_name* is used to search for and trigger a send response within the form file.

The character string you pass in this argument must match a record name or record list name in the form.

send_record

type:	Forms_Record_Data
access:	read only
mechanism:	by reference

Required argument that contains information about the data to be passed from the application program to the form. This argument is a data structure that includes the following:

- Length of the actual record data
- Pointer to the actual record data
- Length of the shadow record data
- Pointer to the shadow record data

The number of send records passed must match the number of records specified in the form for *send_record_name*. This value must be greater than or equal to 0. If the record count is not equal to 1, the count must be passed using the *send_record* request option.

request_options

type:	Forms_Request_Options
access:	read only
mechanism:	by reference

Optional item list containing request-specific arguments that control the environment of the request.

For more information about request options, see Section 6.2.

Table 6.7 contains the Request Option codes that you can specify.

Table 6.7. FORMS_SEND Request Options

Item Code	Description
forms_c_opt_trace	Name of trace file and options
forms_c_opt_completion_status	Location of store completion status
forms_c_opt_completion_routine	Completion routine and parameter
forms_c_opt_end	End of Request Options
forms_c_opt_parent_request	Specifies *parent_request_id*
forms_c_opt_send_record	Number of records being passed
forms_c_opt_send_control	Send control text information
forms_c_opt_timeout	Timeout period

Description

The SEND request causes the Form Manager to pass data from the application program record to form data items. If the form data items to which the Form Manager moves the data are displayed, the Form Manager displays the data on the display device.

Optionally, the SEND RESPONSE for the record or record list can be executed, and various stages of form processing are invoked at the discretion of the form designer.

Examples

1.
```
#define FIRST_NAME_SIZE 15
#define LAST_NAME_SIZE  20
#define CITY_SIZE       20
```

```
⋮


    struct account_str {
         unsigned long   account_number;
         sys_time            date_established;
         unsigned long       zip_code;
         char                last_name[LAST_NAME_SIZE];
         char                first_name[FIRST_NAME_SIZE];
         char                middle_name[15];
         char                street[30];
         char                city[CITY_SIZE];
         char                state[2];
         char                home_phone[10];
         char                work_phone[10];
         char                password[12];
     };


#define ACCOUNT_SIZE    sizeof (struct account_str)

⋮
    Forms_Record_Data record_data; /* Declare record data structure */
    struct account_str      account;
    char* account_name_string = "account";
    record_data.data_record   = &account;
    record_data.data_length   = ACCOUNT_SIZE;
    record_data.shadow_record = NULL;
    record_data.shadow_length = 0;

    status=forms_send (session_id,
                       account_name_string,
                       record_data,
                       )
```

This C example, from the advanced sample checking application, distributes data for the ACCOUNT record to the form. There is no shadow record and no request options.

2.
```
STRUCTURE account
INTEGER balance
END STRUCTURE

RECORD /account/ my_account
RECORD /Forms_Record_Data/ old_account
CHARACTER*16 session_id

my_account.balance = 1000;

old_account.data_record = %LOC(my_account)
old_account.data_length = 4
old_account.shadow_record = 0
old_account.shadow_length = 0

status = forms_send_for(
+                   session_id,
+                   'old_account',
+                   old_account,
+                   0           )
```

This FORTRAN example sends one record named old_account to the form. There is no shadow record and no request options.

RECEIVE

RECEIVE — Copies data from form data items to an application program record.

C Binding

```
#include <formsdef.h>
Forms_Status forms_receive (
                        Forms_Session_Id       session_id,
                        Forms_Text_Ptr         receive_record_name,
                        Forms_Record_Data      receive_record[ ],
                        Forms_Request_Options  request_options[ ] );
```

FORTRAN Binding

```
INCLUDE 'FORMSDEF.F'
INTEGER*4 FUNCTION forms_receive_for ( session_id, receive_record_name,
                                       receive_record, request_options )

CHARACTER*16                      session_id
CHARACTER*(*)                     receive_record_name
RECORD /Forms_Record_Data/        receive_record
RECORD /Forms_Request_Options/    request_options
```

Returns

type:	longword (unsigned)
access:	write only
mechanism:	by value

Arguments

session_id

type:	character-coded text string
access:	read only
mechanism:	by reference

Required argument that contains a unique session-identification string. The Form Manager returns the session-identification string to your application program during the processing of an ENABLE request.

You must pass this argument because the session-identification string identifies which DECforms session and form you want the Form Manager to use during this request.

receive_record_name

type:	character-coded text string

access:	read only
mechanism:	by reference

Required argument that is a character-string name of a form record (for single record transfers) or a form record list (for multiple record transfers). *Receive_record_name* is used to search for a form record or record list to use during data collection and trigger a RECEIVE RESPONSE within the form file.

receive_record

type:	Forms_Record_Data
access:	write only
mechanism:	by reference

Required argument that contains information about the data to be passed from the form to the application program. This argument is a data structure that includes the following:

- Length of the actual record data
- Pointer to the actual record data
- Length of the shadow record data
- Pointer to the shadow record data

The number of receive records passed must match the number of records specified in the form for *receive_record_name*. This value must be greater than 0. If the record count is not equal to 1, it must be passed by using the *receive_record* request option.

request-options

type:	Forms_Request_Options
access:	read only
mechanism:	by reference

Optional list containing request-specific arguments that control the environment of the request.

For more information about request options, see Section 6.2.

Table 6.8 contains the Request Option codes that can be specified and their explanations.

Table 6.8. FORMS_RECEIVE Request Options

Item Code	Description
forms_c_opt_trace	Name of trace file and options
forms_c_opt_completion_status	Location of store completion status
forms_c_opt_completion_routine	Completion routine and parameter
forms_c_opt_end	End of Request Options
forms_c_opt_parent_request	Specifies *parent_request_id*
forms_c_opt_receive_record	Number of records being passed

Item Code	Description
forms_c_opt_receive_control	Receive control text information
forms_c_opt_send_control	Send control text information
forms_c_opt_timeout	Timeout period

Description

The RECEIVE request copies values from form data items to application program record fields. Optionally, the Form Manager executes the RECEIVE RESPONSE for the record or record list.

Example

```
#define print_file_name "new_account.lis"

struct _new_account {
       int balance;
} new_account = {0};

Forms_Request_Options request_option [2];
Forms_Record_Data account_record;
Forms_Session_Id session_id;

account_record.data_length = sizeof (new_account);
account_record.data_record = &new_account;
account_record.shadow_record = NULL;
account_record.shadow_length = 0;

request_option[0].option = forms_c_opt_print;
request_option[0].print.file_name = print_file_name;
request_option[0].print.file_name_length = strlen (print_file_name);

request_option[1].option = forms_c_opt_end;

status = forms_receive(
                    session_id           /* session_id               */
                    "new_account",       /* record name in form      */
                    &account_record,     /* send record structure    */
                    request_option);     /* printfile request option */
```

This C request call receives one record named account_record from the form. There is no shadow record and a print file is specified in Forms_Request_Options.

TRANSCEIVE

TRANSCEIVE — Combines the functionality of the SEND and RECEIVE requests in a single request.

C Binding

```
#include <formsdef.h>
Forms_Status forms_transceive (
                       Forms_Session_Id        session_id,
                       Forms_Text_Ptr          send_record_name,
                       Forms_Record_Data       send_record[ ],
                       Forms_Text_Ptr          receive_record_name,
```

```
                        Forms_Record_Data       receive_record[ ],
                        Forms_Request_Options   request_options[ ] );
```

FORTRAN Binding

```
INCLUDE 'FORMSDEF.F'
INTEGER*4 FUNCTION forms_transceive_for ( session_id, send_record_name,
                                          send_record, receive_record_name,
                                          receive_record, request_options )

CHARACTER*16                        session_id
CHARACTER*(*)                       send_record_name
RECORD /Forms_Record_Data/          send_record
CHARACTER*(*)                       receive_record_name
RECORD /Forms_Record_Data/          receive_record
RECORD /Forms_Request_Options/      request_options
```

Returns

type:	longword (unsigned)
access:	write only
mechanism:	by value

Arguments

See the SEND and RECEIVE sections for descriptions of individual arguments.

Note

On a TRANSCEIVE, the *send_record_count* cannot be 0.

Description

The TRANSCEIVE request causes the Form Manager to combine the functions of a SEND request and a RECEIVE request. First, the Form Manager moves data from application program record fields to form data items. When the request completes, the Form Manager then moves data from form data items to application program record fields.

Example

```
#define print_file_name "new_account.lis"

struct _new_account {
       int balance;
} new_account = {0};

Forms_Request_Options request_option [2];
Forms_Record_Data account_record;
Forms_Session_Id session_id;

account_record.data_record = &new_account;
account_record.data_length = sizeof (new_account);
account_record.shadow_record = NULL;
```

```
account_record.shadow_length = 0;

request_option[0].option = forms_c_opt_print;
request_option[0].print.file_name = print_file_name;
request_option[0].print.file_name_length = strlen (print_file_name);

request_option[1].option = forms_c_opt_end;

status = forms_transceive(
              session_id           /* session_id                   */
              "new_account",       /* send record name in form     */
              &account_record,     /* send record structure        */
              "new_account",       /* receive record name in form  */
              &account_record,     /* receive record structure     */
              request_option);     /* request options              */
```

This C request call transceives one record named account_record from the form. There is no shadow record and a print file is specified in Forms_Request_Options.

CANCEL

CANCEL — Cancels the current request and all outstanding requests for the specified session.

C Binding

```
#include <formsdef.h>
Forms_Status forms_cancel (
                      Forms_Session_Id session_id,
                      Forms_Request_Options request_options[ ] );
```

FORTRAN Binding

```
INCLUDE 'FORMSDEF.F'
INTEGER*4 FUNCTION forms_cancel_for ( session_id, request_options )

CHARACTER*16                          session_id
RECORD /Forms_Request_Options/        request_options
```

Arguments

session_id

type:	character-coded text string
access:	read only
mechanism:	by reference

Required argument that contains a unique session-identification string. The Form Manager returns the session-identification string to your application program during the processing of an ENABLE request.

You must pass this argument because the session-identification string identifies which session you want the Form Manager to cancel.

request_options

type:	Forms_Request_Options
access:	read only
mechanism:	by reference

Optional item list containing request-specific arguments that control the environment of the request.

For more information about request options, see Section 6.2.

Table 6.9 contains the Request Option codes that can be specified and their explanations.

Table 6.9. FORMS_CANCEL Request Options

Item Code	Description
forms_c_opt_completion_status	Location of store completion status
forms_c_opt_completion_routine	Completion routine and parameter
forms_c_opt_end	End of Request Options

Description

The CANCEL request causes the Form Manager to cancel the processing of any request that is in progress for the specified session and any outstanding request for that session. The cancelled requests end with the status forms_s_cancelled.

CANCEL executes asynchronously with respect to the requests that it cancels; CANCEL returns to the routine that called it before it cancels the specified request.

The CANCEL request directs the Form Manager to terminate the current request as quickly as possible. No additional input or output to the screen is started nor is any data returned to the program.

Because a cancelled request may have been outputting data to the screen at the time it was cancelled, the contents of the display may not accurately reflect the current values of form data as stored in the form. To ensure that the screen is correct after you cancel a request, issue the REFRESH response step or press Ctrl/W (on VT devices) to refresh the screen.

The CANCEL request should be restricted to cases where the operator has clear and unmistakable feedback that a request has been cancelled.

Example

```
status = forms_cancel (session_id, NULL );
```

This C example cancels any requests that are in progress and does not pass any request options.

6.8. Structure Definitions for the C Interface

The file included in this section contains the structure definitions for Forms_Request_Options and other predefined types for various DECforms requests. These definitions are part of the C header file, formsdef.h, for the DECforms C request interface.

If you are using the C binding in the portable API to write your application, be sure to include this formsdef.h file.

This listing should only be used as a reference. For the most up-to-date list, refer to the formsdef.h file that is contained in your kit.

```
#if !defined (_FORMS_DEFINED)
#define _FORMS_DEFINED

/*
** ++
**   FACILITY:
**
**       DECforms
**
**   ABSTRACT:
**
**       Include file for the DECforms API.
**
**      NOTE: This file must remain both C and C++ compatible.
**      Do not use C++ style comments or commands in this file.
**
** --
*/

/*
 *        COPYRIGHT (c) 1993, 1997 BY
 *        HEWLETT-PACKARD COMPANY, CALIFORNIA, MASS.
 *
 *  THIS SOFTWARE IS FURNISHED UNDER A LICENSE AND MAY BE USED AND  COPIED
 *  ONLY  IN  ACCORDANCE  WITH  THE  TERMS  OF  SUCH  LICENSE AND WITH THE
 *  INCLUSION OF THE ABOVE COPYRIGHT NOTICE.  THIS SOFTWARE OR  ANY  OTHER
 *  COPIES  THEREOF MAY NOT BE PROVIDED OR OTHERWISE MADE AVAILABLE TO ANY
 *  OTHER PERSON.  NO TITLE TO AND OWNERSHIP OF  THE  SOFTWARE  IS  HEREBY
 *  TRANSFERRED.
 *
 *  THE INFORMATION IN THIS SOFTWARE IS SUBJECT TO CHANGE  WITHOUT  NOTICE
 *  AND  SHOULD  NOT  BE  CONSTRUED  AS  A COMMITMENT BY HP.
 *
 *
 *  HP ASSUMES NO RESPONSIBILITY FOR THE USE OR  RELIABILITY  OF  ITS
 *  SOFTWARE ON EQUIPMENT WHICH IS NOT SUPPLIED BY HP.
 *
*/




/* Possible values for the default terminal type option  */
/* These are used in the Forms_Request_Options, for      */
/* the forms_c_opt_default_term option (below)           */
typedef enum {
               forms_c_dev_vt100       = 0,
               forms_c_dev_vt100_noavo = 1,
               forms_c_dev_vt200       = 2,
               forms_c_dev_vt300       = 3,
               forms_c_dev_vt400       = 4,
               forms_c_dev_mswindows   = 5
             } Forms_Default_Term_Values;

typedef long int Forms_Default_Term_Type;
```

```
/* Possible values for the default color type option     */
/* These are used in the Forms_Request_Options, for      */
/* the forms_c_opt_default_color option (below)          */
typedef enum {
                forms_c_color_mono        = 0,
                forms_c_color_regis       = 4,
                forms_c_color_ansi        = 8
             } Forms_Default_Color_Values;

typedef long int Forms_Default_Color_Type;


/*
 * Define Forms_Form_Object, which points to the global symbol created by
 the commands:
 *      "forms extract object"  (on OpenVms and Ultrix)
 *      "forms object"          (on Windows)
 */
#ifdef vms                                              /* OpenVms     */
#define Forms_Form_Object globalref  void *
#elif  defined (_WIN32) &&  defined (__cplusplus)       /* _WIN32, C++  */
#define Forms_Form_Object extern "C" void *
#elif  defined (_WIN32) && !defined (__cplusplus)       /* _WIN32, C    */
#define Forms_Form_Object extern     void *
#elif  defined (MSDOS) &&  defined (__cplusplus)        /* MSDOS, C++  */
#define Forms_Form_Object extern "C" void __far *
#elif  defined (MSDOS) && !defined (__cplusplus)        /* MSDOS, C    */
#define Forms_Form_Object extern     void __far *
#elif !defined (MSDOS) &&  defined (__cplusplus)        /* Ultrix, C++ */
#define Forms_Form_Object extern "C" void *
#elif !defined (MSDOS) && !defined (__cplusplus)        /* Ultrix, C   */
#define Forms_Form_Object extern     void *
#endif


/*
 * Define Forms_Callback for PEU routine declarations
 *
 *  If your PEU routine is NOT returning a value, you can declare it as:
 *
 * void Forms_Callback peu_routine(...);
 *
 *  If your PEU routine IS returning a value, you can declare it as:
 *
 * long Forms_Callback peu_routine(...);
 */
#if    defined(vms)                                     /* OpenVms     */
#define Forms_Callback
#elif  defined (MSDOS)                                  /* MSDOS       */
#define Forms_Callback __far __pascal __export
#elif  defined (_WIN32)                                 /* WIN32       */
#define Forms_Callback __declspec(dllexport)
#else                                                   /* Ultrix      */
#define Forms_Callback
#endif
```

```
/* Define the pointer and routine linkages - only used internally to
 this .H file */

#if defined (MSDOS)
#define _Mem_Access_ __far
#define _Rtn_Access_ __far __pascal __export
#elif defined (_WIN32)
#define _Mem_Access_
#define _Rtn_Access_
#else
#define _Mem_Access_
#define _Rtn_Access_
#endif

/*
 * Define datatypes which are referenced in the structures, parameters,
 * and calls of the DECforms requests.
 */

typedef long int                Forms_Status;
                                   /* type definition for status of     */
                                   /* a request call                    */
typedef long int _Mem_Access_ * Forms_Status_Ptr;
                                    /* Pointer to Forms_Status return   */
                                    /* value                            */
typedef char                    Forms_Session_Id[16];
                                    /* Session id text string of 16     */
                                    /* characters                       */
typedef char                    Forms_Control_Text_Item[5];
                                    /* Control text item of 5 characters*/
typedef char                    Forms_Control_Text[5][5];
                                    /* Control text array of 5          */
                                    /*   Control_Text_Items             */
typedef Forms_Control_Text _Mem_Access_ * Forms_Control_Text_Ptr;
                                    /* Pointer to            */
                                    /* Forms_Control_Text               */
typedef long int                Forms_Data_Length;
                                    /* Data record  length contained    */
                                    /* in 4 bytes                       */
typedef long int                Forms_Shadow_Length;
                                    /* Shadow record length contained   */
                                    /* in 4 bytes                       */
typedef long int                Forms_Count;
                                    /* Count of objects contained       */
                                    /* in 4 bytes                       */
typedef long int _Mem_Access_ * Forms_Count_Ptr;
                                    /* Pointer to Forms_Count, a 4      */
                                    /* byte value                       */
typedef void     _Mem_Access_ * Forms_Generic_Ptr;
                                    /* General pointer to an object     */
                                    /* contained in 4 bytes             */
typedef long int                Forms_Value;
                                    /*Data value contained in 4         */
typedef long int                Forms_Time_Value;
                                    /* element of ANSI struct           */
                                    /* tm (see time.h)                  */
```

```
typedef long in         Forms_Flags;/* 4 bytes of binary flags value    */
typedef char             Forms_Text;/* Text character                   */
typedef long int                Forms_Text_Length;
                                    /* Count of text characters         */
                                    /* in a string                      */
typedef char     _Mem_Access_ * Forms_Text_Ptr;
                                    /* Pointer to Forms_Text, one or    */
                                    /* more text characters             */
typedef void     _Mem_Access_ * Forms_Routine_Arg_Ptr;
                                    /*Pointer (any type) to Completion  */
                                    /* routine argument                 */
typedef void   (_Rtn_Access_ * Forms_Routine_Ptr)(Forms_Routine_Arg_Ptr);
                                    /* Pointer to completion            */
                                    /* Routine returning Void           */
typedef void     _Mem_Access_ * Forms_Form_Object_Ptr;
                                    /* Pointer to linked in Form file   */
typedef char                    Forms_Request_Id[24];
                                    /* Parent request id text string    */
                                    /* of 24 characters                 */
typedef char     _Mem_Access_ * Forms_Request_Id_Ptr;
                                    /* Pointer to Forms_Request_Id      */

/************************************************************************/
/*                                                                      */
/* Structure for passing send and receive record data and shadow record */
/* data                                                                 */
/*                                                                      */
/************************************************************************/

typedef struct {
Forms_Data_Length  data_length;    /* length of the actual record data  */
                                   /* area                              */
Forms_Generic_Ptr  data_record;    /* pointer to the actual record data */
                                   /* area                              */
Forms_Shadow_Length shadow_length; /* length of the shadow record data  */
                                   /* area                              */
Forms_Generic_Ptr  shadow_record;  /* pointer to the shadow record data */
                                   /* area                              */
} Forms_Record_Data;



/************************************************************************/
/*                                                                      */
/* Structure for building a field in a record message which maps to a   */
/* record field of IFDL data type TM.  ANSI C doesn't say how long the  */
/* tm data type is, only that it contains certain "int" fields.         */
/* The DECforms TM data type is an implementation of the ANSI C data    */
/* type, but your C compiler may have a different (but still            */
/* standard-conforming) definition of TM.                               */
/* Hence this structure can be used to load and unload the record       */
/* message even if your C compiler's implementation of tm contains      */
/* additional fields.                                                   */
/*                                                                      */
/************************************************************************/

typedef struct
   {
```

```
        Forms_Time_Value        tm_sec,  tm_min,  tm_hour;
        Forms_Time_Value        tm_mday, tm_mon,  tm_year;
        Forms_Time_Value        tm_wday, tm_yday, tm_isdst;
   } Forms_Tm;


/* ********************************************************************/
/*                                                                    */
/* Option names for Forms_Request_Options                             */
/*                                                                    */
/********************************************************************/

typedef enum {
 forms_c_opt_selection_label = -10,    /* selection label value         */
 forms_c_opt_default_color = -9,       /* default color type            */
 forms_c_opt_default_term = -8,        /* default terminal type         */
 forms_c_opt_no_term_io = -7,          /* suppresses terminal I/O       */
 forms_c_opt_form = -6,                /* DECforms form Object          */
 forms_c_opt_trace  = -5,              /* name of the trace file and    */
                                       /* options                       */
 forms_c_opt_print = -4,               /* name of printfile to use      */
 forms_c_opt_language = -3,            /* name of language to match     */
 forms_c_opt_completion_status = -2,   /* location to store completion  */
                                       /* status                        */
 forms_c_opt_completion_routine = -1,  /* completion routine and        */
                                       /* parameter                     */
 forms_c_opt_end = 0,                  /* indicates end of request      */
                                       /* options                       */
 forms_c_opt_parent_request = 1,       /* specifies parent request id   */
 forms_c_opt_receive_record = 2,       /* number of records being       */
                                       /* passed                        */
 forms_c_opt_send_record = 3,          /* number of records being       */
                                       /* passed                        */
 forms_c_opt_receive_control = 4,      /* receive control text info     */
 forms_c_opt_send_control = 5,         /* send control text info        */
 forms_c_opt_timeout = 6               /* describes timeout period      */
             } Forms_Request_Options_Values;

typedef long int Forms_Request_Options_Type;


/********************************************************************/
/*                                                                  */
/* Forms_Request_Options for API request calls                      */
/*                                                                  */
/********************************************************************/


typedef union {

 /* Discriminate Tag */

 Forms_Request_Options_Type option;

 struct {                            /* forms_c_opt_selection_label */
    Forms_Request_Options_Type option;
    Forms_Text_Ptr    name;          /* Name of layout to choose */
    Forms_Text_Length name_length    /* length of name           */
```

```
} selection_label;

struct {                            /* forms_c_opt_default_color */
   Forms_Request_Options_Type option;
   Forms_Default_Color_Type type;   /* when runtime can't detect      */
                                    /* whether                        */
                                    /* you have color, set this value */
} default_color;

struct {                            /* forms_c_opt_default_term */
   Forms_Request_Options_Type option;
   Forms_Default_Term_Type type;    /* when runtime can't detect the */
                                    /* terminal type, set this value */
} default_term;

struct {                            /* forms_c_opt_no_term_io */
   Forms_Request_Options_Type option;
   Forms_Flags flag;
} no_term_io;

struct {                            /* forms_c_opt_form */
   Forms_Request_Options_Type option;
   Forms_Form_Object_Ptr object;    /* Pointer created by             */
                                    /* Forms_Form_Object,             */
                                    /* in application program         */
} form;

struct {                            /* forms_c_opt_trace */
   Forms_Request_Options_Type option;
   Forms_Text_Ptr    file_name;     /* Name of trace file */
   Forms_Text_Length file_name_length;  /* length of name     */
   Forms_Flags flag;
} trace;

struct {                            /* forms_c_opt_print */
   Forms_Request_Options_Type option;
   Forms_Text_Ptr    file_name;     /* Name of file or device that */
                                    /* the PRINT step uses */
   Forms_Text_Length file_name_length; /* length of name      */
} print;

struct {                            /* forms_c_opt_language */
   Forms_Request_Options_Type option;
   Forms_Text_Ptr    name;          /* Name to use when selecting a */
                                    /* layout, matches the LANGUAGE */
                                    /* clause                       */
   Forms_Text_Length name_length;   /* length of name               */
} language;

struct {                            /* forms_c_opt_completion_status */
   Forms_Request_Options_Type option;
   Forms_Status_Ptr address;        /* Address to load completion */
                                    /* status when request completes */
} completion_status;

struct {                            /* forms_c_opt_completion_routine */
   Forms_Request_Options_Type option;
   Forms_Routine_Ptr     address;   /* Address of routine to call when */
```

```
                                    /* request completes  */
   Forms_Routine_Arg_Ptr parameter; /* Address of parameter to pass */
                                    /* to completion routine */
} completion_routine;




struct {                            /* forms_c_opt_end */
   Forms_Request_Options_Type option;
} end;

struct {                            /* forms_c_opt_parent_request */
   Forms_Request_Options_Type option;
   Forms_Request_Id_Ptr id;         /* Pointer to parent request id,   */
                                    /* for                             */
                                    /* calls to DECforms while a       */
} parent_request;                   /* DECforms request is still active */
                                    /* (such as from an Escape Routine) */

struct {                            /* forms_c_opt_receive_record */
   Forms_Request_Options_Type option;
   Forms_Count count;
} receive_record;         /* Number of unique receive records in the   */
                          /* request                                   */


struct {                            /* forms_c_opt_send_record */
   Forms_Request_Options_Type option;
   Forms_Count count;
} send_record;            /* Number of unique send records in the      */
                          /* request                                   */


struct {                            /* forms_c_opt_receive_control */
   Forms_Request_Options_Type option;
   Forms_Count_Ptr text_count;
                     /* Pointer to Number of control text items        */
                     /* returned by                                    */
                                    /* DECforms - write only */
   Forms_Control_Text_Ptr text;
                     /* Receive control text must be an array of 25    */
                     /* bytes                                          */
} receive_control;

struct {                            /* forms_c_opt_send_control */
   Forms_Request_Options_Type option;
   Forms_Count text_count;          /* Number of control text being    */
                                    /* sent to DECforms                */
                                    /* - read only    */
   Forms_Control_Text_Ptr text;
                           /* Send control text could be up to 25     */
                           /* bytes, in multiples of 5                */
} send_control;

struct {                            /* forms_c_opt_timeout */
   Forms_Request_Options_Type option;
   Forms_Value period;
```

```
                                /* Timeout value is limited to the      */
                                /* range 0 to 32767 seconds on VMS.     */
 } timeout;

 struct {
    Forms_Request_Options_Type option;
                                     /*This structure is placed here to */
                                     /* ensure extensibility.   */
    char forms_reserved_bytes[28];
                                   /* Please do not use this structure. */
 } forms_reserved_struct;


} Forms_Request_Options;

/*                                                                      */
/* FIMS specified status value                                          */
/*                                                                      */
/* Note: S000 and ESnnn are corresponding FIMS control texts.           */
/*                                                                      */
/*       Severity of each status is indicated by:                       */
/*                                                                      */
/* (E) - Fatal error(s) occurred during request. Must be fixed before  */
/*       continuing.                                                    */
/* (W) - Warning error(s) occurred during request. Should fix before   */
/*       continuing.                                                    */
/* (I) - Informational event(s) occurred during request.                */
/* (S) - Request processed successfully.                                */
/*                                                                      */

typedef enum {
 forms_s_normal             =  0,
                 /*  S000: (S) Request calls completed successfully    */
 forms_s_timeout            =  1,
                 /* ES001: (E) Input did not complete in the time      */
                 /*            specified                               */
 forms_s_formerror          =  2,
                 /* ES002: (E) Encountered problem when using form file */
 forms_s_nolayout           =  3,
                 /* ES003: (E) No layout fit terminal type, language and */
                 /*            display size                            */
 forms_s_invdevice           =  4,
                 /* ES004: (E) Invalid device specified in ENABLE      */
 forms_s_hangup              =  5,
                 /* ES005: (E) Data set hangup; session disabled       */
 forms_s_norecord            =  7,
                 /* ES007: (E) Specified record identifier not in form */
 forms_s_badreclen           =  8,
                 /* ES008: (E) Record length argument did not match    */
                 /*            length                                  *
                 /*            of record in form                       */
 forms_s_inuse               = 10,
                 /* ES010: (E) Attempted to disable a form still in    */
                 /*            use                                     */
 forms_s_nosession           = 11,
                 /* ES011: (E) Session id in argument did not match    */
                 /*            existing session                        */
 forms_s_return_immed        = 12,
```

```
                /* ES012: (S) Request completed due to REQUEST        */
                /*            IMMEDIATE                               */
 forms_s_nodecpt            = 14,
                /* ES014: (E) Decimal or comma decimal point          */
                /*            positioned incorrectly in record field  */
 forms_s_bad_rshdwreclen    = 15,
                /* ES015: (E) Receive-shadow-record-length did not    */
                /*            match length specified in form          */
 forms_s_bad_sshdwreclen    = 16,
                /* ES016: (E) Send-shadow-record-length is something  */
                /*            other than 1                            */
 forms_s_cancelled          = 17,
                /* ES017: (E) Request interrupted by arrival of CANCEL */
 forms_s_noactreq           = 19,
                /* ES019: (E) No active requests to CANCEL            */
 forms_s_invlobound         = 24,
                /* ES024: (E) Subscript reference less than base      */
 forms_s_invhibound         = 25,
                /* ES025: (E) Subscript reference greater than array  */
                /*            dimension defined                       */


/*                                                                    */
/* DECforms specific status values                                    */
/*                                                                    */
/* Note: EInnn are corresponding DECforms control texts.              */
/*       Any status relating to parameter checking of DECforms request */
/*       calls or license checking of DECforms will not have a        */
/*       corresponding control text.                                  */
/*                                                                    */
/*       Severity of each status is indicated by:                     */
/*                                                                    */
/*       (E) - Fatal error(s) occurred during request.                */
/*             Must be fixed before continuing.                       */
/*       (W) - Warning error(s) occurred during request.              */
/*             Should fix before continuing.                          */
/*       (I) - Informational event(s) occurred during request.        */
/*       (S) - Request processed successfully.                        */
/*                                                                    */

    forms_s_illdtcvt           =  -1,
                    /* EI001: (W) Illegal DATE, TIME, ADT conversion   */
    forms_s_badreccnt          =  -2,
                    /* EI002: (E) number of records did not match      */
                    /*             number specified in record list     */
    forms_s_converr            =  -3,
                    /* EI003: (I) Error while converting from one data */
                    /*            type to another                      */
    forms_s_aborted            =  -4,
                    /* EI004: (E) Session aborted due to severe error  */
                    /*            in another request                   */
    forms_s_badarg             =  -5,
                    /*      : (E) Bad argument or incorrect format     */
    forms_s_baditmlstcode      =  -6,
                    /*      : (E) Invalid item code found in item list */
    forms_s_baditmlstcomb      =  -7,
                    /*      : (E) Invalid combination of item codes    */
```

```
                /*            found in item list                  */
forms_s_blocked_by_ast     =  -8,
                /* EI008: (E) Cannot process request; block by     */
                /*            application AST                     */
forms_s_bad_devhlr         =  -9,
                /* EI009: (E) Invalid device handler               */
forms_s_caninprog           = -10,
                /* EI010: (E) A previous CANCEL is still in progress */
forms_s_closetrace         = -11,
                /* EI011: (W) Cannot close trace file              */
forms_s_deverr             = -12,
                /* EI012: (E) Device I/O error                     */
forms_s_disinprog          = -13,
                /* EI013: (E) A previous DISABLE is still in       */
                /*            progress                             */
forms_s_exprevalerr        = -14,
                /* EI014: (E) Cannot convert operands into common  */
                /*            data type                            */
forms_s_fatinterr          = -15,
                /* EI015: (E) Fatal Internal error                 */
forms_s_illctltxtcnt       = -16,
                /*      : (E) Illegal control text count argument  */
forms_s_illfldvaluectx     = -17,
                /* EI017: (E) Illegal FIELDVALUE context           */
forms_s_illvpuse           = -18,
                /* EI018: (E) Illegal use of print viewport        */
forms_s_intdatcor          = -19,
                /* EI019: (E) Database consistency check failed    */
forms_s_invrange           = -20,
                /* EI020: (E) Invalid subscript range              */
forms_s_invreccnt          = -21,
                /* EI021: (E) Invalid record count value           */
forms_s_invrecdes          = -22,
                /* EI022: (E) Invalid record message descriptor    */
forms_s_nohandler          = -23,
                /* EI023: (E) No device handler for such device    */
forms_s_nolicense          = -24,
                /*      : (E) No DECforms software license is active */
forms_s_noparent           = -25,
                /* EI025: (E) Specified parent request did not exist */
forms_s_no_read_access     = -26,
                /*      : (E) No read access to user argument      */
forms_s_openout            = -27,
                /* EI027: (E) The specified output file cannot be  */
                /*            opened                               */
forms_s_opentrace          = -28,
                /* EI028: (W) Cannot open trace file for output    */
forms_s_paramovrflow       = -29,
                /* EI029: (E) An escape routine parameter has      */
                /*            overflowed                           */
forms_s_procesc_not_found  = -30,
                /* EI030: (E) Address of procedural escape not found */
forms_s_proc_escape_error  = -31,
                /* EI031: (E) Request terminated due to severe error */
                /*            in PEU                                */
forms_s_recvrecitems       = -32,
                /*      : (E) # of receive record items did not    */
                /*            match record count value             */
```

```
    forms_s_reqdarg            = -33,
                    /*      : (E) Required argument missing            */
    forms_s_sendrecitems       = -34,
                    /*      : (E) # of send record items did not match */
                    /*            record count value                   */
    forms_s_strtooshort        = -35,
                    /* EI035: (E) length of string is too small        */
    forms_s_writetrace         = -36,
                    /* EI036: (W) Cannot write to trace file           */
    forms_s_no_write_access    = -37,
                    /*      : (E) No write access to user argument     */
    forms_s_bckgrndio          = -38,
                    /* EI038: (E) Attempted read or write I/O from     */
                    /*            background process                   */
    forms_s_timeract           = -39,
                    /* EI039: (E) Attempted timed field input while    */
                    /*            alarm active                         */
    forms_s_blkbyreq           = -40,
                    /* EI040: (E) Attempted synchronous request while  */
                    /*            another request active               */
    forms_s_imgmismatch        = -41,
                    /*      : (E) Shareable image mismatch             */
    forms_s_nyi                = -42,
                    /* EI042: (E) Requested operation is not yet       */
                    /*            implemented                          */
    forms_s_cantopendic        = -46
                   /* EI046: (E) Can't open the Kana-Kanji conversion  */
                   /*            dictionary                            */

} Forms_Status_Values;


/* Prevent the name mangling that C++ does, so that we keep these      */
/* external names                                                      */
#if defined (__cplusplus)
extern "C"
{
#endif


/***********************************************************************/
/*                                                                     */
/* Define the function prototypes for the 6 DECforms calls             */
/*                                                                     */
/* These are the definitions for those compilers that DO               */
/* support function prototypes with parameter definitions              */
/*                                                                     */
/***********************************************************************/

#if defined(__STDC__) || defined (vaxc) || defined (MSDOS) || defined
 (__cplusplus)

Forms_Status _Rtn_Access_ forms_enable
                (char                  _Mem_Access_ * session_id,
                 Forms_Text_Ptr                       device_name,
                 Forms_Text_Ptr                       form_file_name,
                 Forms_Text_Ptr                       form_name,
                 Forms_Request_Options _Mem_Access_   request_options[]);
```

```
Forms_Status _Rtn_Access_ forms_send
                (char                  _Mem_Access_ * session_id,
                 Forms_Text_Ptr                       send_record_name,
                 Forms_Record_Data     _Mem_Access_   send_record[],
                 Forms_Request_Options _Mem_Access_   request_options[]);

Forms_Status _Rtn_Access_ forms_receive
                (char                  _Mem_Access_ * session_id,
                 Forms_Text_Ptr                       receive_record_name,
                 Forms_Record_Data     _Mem_Access_   receive_record[],
                 Forms_Request_Options _Mem_Access_   request_options[]);

Forms_Status _Rtn_Access_ forms_transceive
                (char                  _Mem_Access_ * session_id,
                 Forms_Text_Ptr                       send_record_name,
                 Forms_Record_Data     _Mem_Access_   send_record[],
                 Forms_Text_Ptr                       receive_record_name,
                 Forms_Record_Data     _Mem_Access_   receive_record[],
                 Forms_Request_Options _Mem_Access_   request_options[]);

Forms_Status _Rtn_Access_ forms_disable
                (char                  _Mem_Access_ * session_id,
                 Forms_Request_Options _Mem_Access_   request_options[]);

Forms_Status _Rtn_Access_ forms_cancel
                (char                  _Mem_Access_ * session_id,
                 Forms_Request_Options _Mem_Access_   request_options[]);

#else



/************************************************************************/
/*                                                                      */
/* Define the function prototypes for the 6 DECforms calls              */
/*                                                                      */
/* These are the definitions for those compilers that do NOT            */
/* support function prototypes with parameter definitions               */
/*                                                                      */
/************************************************************************/

Forms_Status forms_enable();
Forms_Status forms_send();
Forms_Status forms_receive();
Forms_Status forms_transceive();
Forms_Status forms_disable();
Forms_Status forms_cancel();

#endif         /* End of "if defined STDC, etc" */

/* End of "Prevent the name mangling that C++ does, so that we keep      */
/* these external names"                                                 */
#if defined (__cplusplus)
}
#endif
```

```
#undef _Rtn_Access_
#undef _Mem_Access_

#endif         /* if !defined _FORMS_DEFINED */
```

6.9. Structure Definitions for the Portable API FORTRAN Interface

The file included in this section contains the structure definitions for Forms_Record_Data, as well as the name tag constants and union/map structure declaration for Forms_Request_Options. These declarations are part of the FORTRAN file, formsdef.f, for the DECforms FORTRAN portable binding.

If you are using the FORTRAN binding in the portable API for your program, be sure to include the formsdef.f file in your subprograms and main program.

This listing should only be used as a reference. For the most up-to-date information, refer to the formsdef.f file that ships in your kit.

```
C
C  ++
C    FACILITY:
C
C        DECforms
C
C    ABSTRACT:
C
C        Include file for the DECforms API.
C
C  --
C

C ++
C
C   structure for passing send and receive record data and shadow record
C   data
C
C --
        STRUCTURE /Forms_Record_Data/
          integer      data_length
          integer      data_record
          integer      shadow_length
          integer      shadow_record
        END STRUCTURE

C  ++
C
C       Declaration for literals
C
C  --
        INTEGER forms_c_dev_vt100
        INTEGER forms_c_dev_vt100_noavo
        INTEGER forms_c_dev_vt200
        INTEGER forms_c_dev_vt300
        INTEGER forms_c_dev_vt400
        INTEGER forms_c_color_mono
```

```
        INTEGER forms_c_color_regis
        INTEGER forms_c_color_ansi

C  ++
C       Possible value for the default terminal type option.
C  --
        PARAMETER (forms_c_dev_vt100 = 0)
        PARAMETER (forms_c_dev_vt100_noavo = 1)
        PARAMETER (forms_c_dev_vt200 = 2)
        PARAMETER (forms_c_dev_vt300 = 3)
        PARAMETER (forms_c_dev_vt400 = 4)

C  ++
C       Possible value for the default color type option.
C  --
        PARAMETER (forms_c_color_mono = 0)
        PARAMETER (forms_c_color_regis = 4)
        PARAMETER (forms_c_color_ansi = 8)

C  ++
C
C     Type declaration for option names
C
C  --
        INTEGER forms_c_opt_selection_label
        INTEGER forms_c_opt_default_color
        INTEGER forms_c_opt_default_term
        INTEGER forms_c_opt_no_term_io
        INTEGER forms_c_opt_form
        INTEGER forms_c_opt_trace
        INTEGER forms_c_opt_print
        INTEGER forms_c_opt_language
        INTEGER forms_c_opt_completion_status
        INTEGER forms_c_opt_completion_routine
        INTEGER forms_c_opt_end
        INTEGER forms_c_opt_parent_request
        INTEGER forms_c_opt_receive_record
        INTEGER forms_c_opt_send_record
        INTEGER forms_c_opt_receive_control
        INTEGER forms_c_opt_send_control
        INTEGER forms_c_opt_timeout
C  ++
C
C     Option names for Forms_Request_Options
C
C  --
        PARAMETER (forms_c_opt_selection_label = -10)
        PARAMETER (forms_c_opt_default_color = -9)
        PARAMETER (forms_c_opt_default_term = -8)
        PARAMETER (forms_c_opt_no_term_io = -7)
        PARAMETER (forms_c_opt_form  = -6)
        PARAMETER (forms_c_opt_trace = -5)
        PARAMETER (forms_c_opt_print = -4)
        PARAMETER (forms_c_opt_language = -3)
        PARAMETER (forms_c_opt_completion_status = -2)
        PARAMETER (forms_c_opt_completion_routine = -1)
        PARAMETER (forms_c_opt_end = 0)
        PARAMETER (forms_c_opt_parent_request = 1)
```

```
        PARAMETER (forms_c_opt_receive_record = 2)
        PARAMETER (forms_c_opt_send_record = 3)
        PARAMETER (forms_c_opt_receive_control = 4)
        PARAMETER (forms_c_opt_send_control = 5)
        PARAMETER (forms_c_opt_timeout = 6)


C ++
C
C     Forms_Request_Options for API request calls
C
C --

         STRUCTURE /Forms_Request_Options/

           integer option

           UNION
C
C            forms_c_opt_selection_label
C
             MAP
               integer selection_label
               integer selection_label_length
             END MAP
C
C            forms_c_opt_default_color

 (forms_c_color_mono,forms_c_color_regis,forms_c_color_ansi)
C
             MAP
               integer  default_color_type
             END MAP
C
C            forms_c_opt_default_term   i.e.
                     forms_c_dev_vt100, forms_c_dev_vt200, etc.
C
             MAP
               integer  default_term_type
             END MAP
C
C            forms_c_opt_no_term_io
C
             MAP
               integer  no_term_io_flag
             END MAP
C
C            forms_c_opt_form
C
             MAP
               integer form_object
             END MAP
C
C            forms_c_opt_trace
C
             MAP
               integer  trace_file_name
               integer  trace_file_name_length
```

```
              integer  trace_flag
            END MAP
C
C           forms_c_opt_print
C
            MAP
              integer print_file_name
              integer print_file_name_length
            END MAP
C
C           forms_c_opt_language
C
            MAP
              integer language_name
              integer language_name_length
            END MAP
C
C           forms_c_opt_completion_status
C
            MAP
              integer completion_status_address
            END MAP
C
C           forms_c_opt_completion_routine
C
            MAP
              integer completion_routine_address
              integer completion_routine_parameter
            END MAP
C
C           forms_c_opt_parent_request
C
            MAP
              integer parent_request_id
            END MAP
C
C           forms_c_opt_receive_record
C
            MAP
              integer receive_record_count
            END MAP
C
C           forms_c_opt_send_record
C
            MAP
              integer send_record_count
            END MAP
C
C           forms_c_opt_receive_control
C
C           Number of control text returned by DECforms - write only
C           receive control text must be an array of 25 bytes
C
            MAP
              integer receive_control_text_count
              integer receive_control_text
            END MAP
C
```

```
C            forms_c_opt_send_control
C
C            Number of control text being sent to DECforms - read only
C            send control text could be up to 25 bytes
C
             MAP
               integer send_control_text_count
               integer send_control_text
             END MAP


C
C            forms_c_opt_timeout
C
C            timeout value is limited to the range between 0 to 32767
C            seconds on VMS.
C
             MAP
               integer timeout_period
             END MAP
C
C            This structure is placed here to ensure extensibility.
C            Please do not use this structure.
C
             MAP
               CHARACTER*28 forms_reserved_bytes
             END MAP

           END UNION
        END STRUCTURE
C  ++
C
C     Type declaration for error numbers
C
C  --

        INTEGER forms_s_normal
        INTEGER forms_s_timeout
        INTEGER forms_s_formerror
        INTEGER forms_s_nolayout
        INTEGER forms_s_invdevice
        INTEGER forms_s_hangup
        INTEGER forms_s_norecord
        INTEGER forms_s_badreclen
        INTEGER forms_s_inuse
        INTEGER forms_s_nosession
        INTEGER forms_s_return_immed
        INTEGER forms_s_nodecpt
        INTEGER forms_s_bad_rshdwreclen
        INTEGER forms_s_bad_sshdwreclen
        INTEGER forms_s_cancelled
        INTEGER forms_s_noactreq
        INTEGER forms_s_invlobound
        INTEGER forms_s_invhibound
        INTEGER forms_s_illdtcvt
        INTEGER forms_s_badreccnt
        INTEGER forms_s_converr
        INTEGER forms_s_aborted
```

```
        INTEGER forms_s_badarg
        INTEGER forms_s_baditmlstcode
        INTEGER forms_s_baditmlstcomb
        INTEGER forms_s_blocked_by_ast
        INTEGER forms_s_bad_devhlr
        INTEGER forms_s_caninprog
        INTEGER forms_s_closetrace
        INTEGER forms_s_deverr
        INTEGER forms_s_disinprog
        INTEGER forms_s_exprevalerr
        INTEGER forms_s_fatinterr
        INTEGER forms_s_illctltxtcnt
        INTEGER forms_s_illfldvaluectx
        INTEGER forms_s_illvpuse
        INTEGER forms_s_intdatcor
        INTEGER forms_s_invrange
        INTEGER forms_s_invreccnt
        INTEGER forms_s_invrecdes
        INTEGER forms_s_nohandler
        INTEGER forms_s_nolicense
        INTEGER forms_s_noparent
        INTEGER forms_s_no_read_access
        INTEGER forms_s_openout
        INTEGER forms_s_opentrace
        INTEGER forms_s_paramovrflow
        INTEGER forms_s_procesc_not_found
        INTEGER forms_s_proc_escape_error
        INTEGER forms_s_recvrecitems
        INTEGER forms_s_reqdarg
        INTEGER forms_s_sendrecitems
        INTEGER forms_s_strtooshort
        INTEGER forms_s_writetrace
        INTEGER forms_s_no_write_access
        INTEGER forms_s_bckgrndio
        INTEGER forms_s_timeract
        INTEGER forms_s_blkbyreq
        INTEGER forms_s_imgmismatch
        INTEGER forms_s_nyi
        INTEGER forms_s_cantopendic


CC ++
CC
CC  FIMS specified status value
CC
CC  Note: S000 and ESnnn are corresponding FIMS control texts.
CC
CC        Severity of each status is indicated by:
CC
CC        (E) - Fatal error(s) occurred during request. Must be fixed
CC              before continuing
CC        (W) - Warning error(s) occurred during request. Should fix
CC              before continuing.
CC        (I) - Informational event(s) occurred during request.
CC        (S) - Request processed successfully.
CC --
C
C   S000: (S) Request calls completed successfully
```

```
C
          PARAMETER (forms_s_normal = 0000)
C
C  ES001: (E) Input did not complete in the time specified
C
          PARAMETER (forms_s_timeout = 0001)
C
C  ES002: (E) Encountered problem when using form file
C
          PARAMETER (forms_s_formerror = 0002)
C
C  ES003: (E) No layout fit terminal type, language and display size
C
          PARAMETER (forms_s_nolayout = 0003)
C
C  ES004: (E) Invalid device specified in ENABLE
C
          PARAMETER (forms_s_invdevice = 0004)
C
C  ES005: (E) Data set hangup; session disabled
C
          PARAMETER (forms_s_hangup = 0005)
C
C  ES007: (E) Specified record identifier not in form
C
          PARAMETER (forms_s_norecord = 0007)
C
C  ES008: (E) Record length argument not match length of record in form
C
          PARAMETER (forms_s_badreclen = 0008)
C
C  ES010: (E) Attempted to disable a form still in use
C
          PARAMETER (forms_s_inuse = 0010)
C
C  ES011: (E) Session id in argument not match existing session
C
          PARAMETER (forms_s_nosession = 0011)
C
C  ES012: (S) Request completed due to REQUEST IMMEDIATE
C
          PARAMETER (forms_s_return_immed = 0012)
C
C  ES014: (E) Decimal or comma decimal point positioned incorrectly in
C             record field
C
          PARAMETER (forms_s_nodecpt = 0014)
C
C  ES015: (E) Receive shadow record length not match length specified
C             in form
C
          PARAMETER (forms_s_bad_rshdwreclen = 0015)
C
C  ES016: (E) Send-shadow-record-length is something other than 1
C
          PARAMETER (forms_s_bad_sshdwreclen = 0016)
C
C  ES017: (E) Request interrupted by arrival of CANCEL
```

```
C
          PARAMETER (forms_s_cancelled = 0017)
C
C  ES019: (E) No active requests to CANCEL
C
          PARAMETER (forms_s_noactreq = 0019)
C
C  ES024: (E) Subscript reference less than base
C
          PARAMETER (forms_s_invlobound = 0024)
C
C  ES025: (E) Subscript reference greater than array dimension defined
C
          PARAMETER (forms_s_invhibound = 0025)


CC ++
CC
CC  DECforms specific status values
CC
CC  Note: EInnn are corresponding DECforms control texts. Any status
CC        relating to parameter checking of DECforms request calls
CC        or license checking of DECforms will have not a
CC        corresponding control text.
CC
CC        Severity of each status is indicated by:
CC
CC        (E) - Fatal error(s) occurred during request. Must be
CC              fixed before continuing.
CC        (W) - Warning error(s) occurred during request. Should
CC              fix before continuing.
CC        (I) - Informational event(s) occurred during request.
CC        (S) - Request processed successfully.
CC
CC --
C
C  EI001: (W) Illegal DATE, TIME, ADT conversion
C
          PARAMETER (forms_s_illdtcvt = -0001)
C
C  EI002: (E) number of records not match number specified in
C             record list
C
          PARAMETER (forms_s_badreccnt = -0002)
C
C  EI003: (I) Error while converting from one data type to
C             another
C
          PARAMETER (forms_s_converr = -0003)
C
C  EI004: (E) Session aborted due to severe error in another
C             request
C
          PARAMETER (forms_s_aborted = -0004)
C
C       : (E) Bad argument or incorrect format
C
          PARAMETER (forms_s_badarg = -0005)
```

```
C
C       : (E) Invalid item code found in item list
C
          PARAMETER (forms_s_baditmlstcode = -0006)
C
C       : (E) Invalid combination of item codes found in
C             item list
C
          PARAMETER (forms_s_baditmlstcomb = -0007)
C
C  EI008: (E) Cannot process request; block by application
C             AST
C
          PARAMETER (forms_s_blocked_by_ast = -0008)
C
C  EI009: (E)Invalid device handler
C
          PARAMETER (forms_s_bad_devhlr = -0009)
C
C  EI010: (E) A previous CANCEL is still in progress
C
          PARAMETER (forms_s_caninprog = -0010)
C
C  EI011: (W) Cannot close trace file
C
          PARAMETER (forms_s_closetrace = -0011)
C
C  EI012: (E) Device I/O error
C
          PARAMETER (forms_s_deverr = -0012)
C
C  EI013: (E) A previous DISABLE is still in progress
C
          PARAMETER (forms_s_disinprog = -0013)
C
C  EI014: (E) Cannot convert operands into common data type
C
          PARAMETER (forms_s_exprevalerr = -0014)
C
C  EI015: (E) Fatal Internal error
C
          PARAMETER (forms_s_fatinterr = -0015)
C
C       : (E) Illegal control text count argument
C
          PARAMETER (forms_s_illctltxtcnt = -0016)
C
C  EI017: (E) Illegal FIELDVALUE context
C
          PARAMETER (forms_s_illfldvaluectx = -0017)
C
C  EI018: (E) Illegal use of print viewport
C
          PARAMETER (forms_s_illvpuse = -0018)
C
C  EI019: (E) Database consistency check failed
C
          PARAMETER (forms_s_intdatcor = -0019)
```

```
C
C  EI020: (E) Invalid subscript range
C
          PARAMETER (forms_s_invrange = -0020)
C
C  EI021: (E) Invalid record count value
C
          PARAMETER (forms_s_invreccnt = -0021)
C
C  EI022: (E) Invalid record message descriptor
C
          PARAMETER (forms_s_invrecdes = -0022)
C
C  EI023: (E) No device handler for such device
C
          PARAMETER (forms_s_nohandler = -0023)


C
C       : (E) No DECforms software license is active
C
          PARAMETER (forms_s_nolicense = -0024)
C
C  EI025: (E) Specified parent request not exist
C
          PARAMETER (forms_s_noparent = -0025)
C
C       : (E) No read access to user argument
C
          PARAMETER (forms_s_no_read_access = -0026)
C
C  EI027: (E) The specified output file cannot be opened
C
          PARAMETER (forms_s_openout = -0027)
C
C  EI028: (W) Cannot open trace file for output
C
          PARAMETER (forms_s_opentrace = -0028)
C
C  EI029: (E) An escape routine parameter has overflowed
C
          PARAMETER (forms_s_paramovrflow = -0029)
C
C  EI030: (E) Address of procedural escape not found
C
          PARAMETER (forms_s_procesc_not_found = -0030)
C
C  EI031: (E) Request terminated due to severe error in PEU
C
          PARAMETER (forms_s_proc_escape_error = -0031)
C
C       : (E) # of receive record items not match record count value
C
          PARAMETER (forms_s_recvrecitems = -0032)
C
C       : (E) Required argument missing
C
          PARAMETER (forms_s_reqdarg = -0033)
```

```
C
C       : (E) # of send record items not match record count value
C
          PARAMETER (forms_s_sendrecitems = -0034)
C
C  EI035: (E) length of string is too small
C
          PARAMETER (forms_s_strtooshort = -0035)
C
C  EI036: (W) Cannot write to trace file
C
          PARAMETER (forms_s_writetrace = -0036)
C
C       : (E) No write access to user argument
C
          PARAMETER (forms_s_no_write_access = -0037)
C
C  EI038: (E) Attempted read or write I/O from background process
C
          PARAMETER (forms_s_bckgrndio = -0038)
C
C  EI039: (E) Attempted timed field input while alarm active
C
          PARAMETER (forms_s_timeract = -0039)
C
C  EI040: (E) Attempted synchronous request while another request active
C
          PARAMETER (forms_s_blkbyreq = -0040)
C
C       : (E) Shareable image mismatch
C
          PARAMETER (forms_s_imgmismatch = -0041)
C
C  EI042: (E) Requested operation is not yet implemented
C
          PARAMETER (forms_s_nyi = -0042)
C
C  EI046: (E) Can't open the Kana-Kanji dictionary
C
          PARAMETER (forms_s_cantopendic = -0046)


C
C    DECforms F77 entry points
C
C    forms_status = forms_enable_for ( session_id, device_name,
C                   form_file_name, form_name, request_options )
C
C                   CHARACTER*16  session_id
C                   CHARACTER*(*) device_name
C                   CHARACTER*(*) form_file_name
C                   CHARACTER*(*) form_name
C                   RECORD /Forms_Request_Options/ request_options()
C
C    forms_status = forms_send_for ( session_id, send_record_name,
C                                    send_record, request_options )
C
C                   CHARACTER*16  session_id
```

```
C                   CHARACTER*(*) send_record_name
C                   RECORD /Forms_Record_Data/ send_record()
C                   RECORD /Forms_Request_Options/ request_options()
C
C    forms_status = forms_receive_for ( session_id, receive_record_name,
C                                       receive_record, request_options )
C
C                   CHARACTER*16  session_id
C                   CHARACTER*(*) receive_record_name
C                   RECORD /Forms_Record_Data/ receive_record()
C                   RECORD /Forms_Request_Options/ request_options()
C
C    forms_status = forms_transceive_for ( session_id, send_record_name,
C                                          send_record,
C                                          receive_record_name,
C                                          receive_record,
C                                          request_options )
C
C                   CHARACTER*16  session_id
C                   CHARACTER*(*) send_record_name
C                   RECORD /Forms_Record_Data/ send_record()
C                   CHARACTER*(*) receive_record_name
C                   RECORD /Forms_Record_Data/ receive_record()
C                   RECORD /Forms_Request_Options/ request_options()
C
C    forms_status = forms_disable_for ( session_id, request_options )
C
C                   CHARACTER*16  session_id
C                   RECORD /Forms_Request_Options/ request_options()
C
C    forms_status = forms_cancel_for ( session_id, request_options )
C
C                   CHARACTER*16  session_id
C                   RECORD /Forms_Request_Options/ request_options()
C
C

        INTEGER   forms_enable_for
        EXTERNAL  forms_enable_for

        INTEGER   forms_send_for
        EXTERNAL  forms_send_for

        INTEGER   forms_receive_for
        EXTERNAL  forms_receive_for

        INTEGER   forms_transceive_for
        EXTERNAL  forms_transceive_for

        INTEGER   forms_disable_for
        EXTERNAL  forms_disable_for

        INTEGER   forms_cancel_for
        EXTERNAL  forms_cancel_for
```

Appendix A. Elementary Conditions

Elementary conditions are predefined accept phase conditions. During accept phase processing, the Form Manager stores either true or false in elementary conditions to indicate the status of activation list processing. You can use elementary conditions in conditional expressions in your IFDL source file.

The following table explains what state causes each of the elementary conditions to be true.

Elementary Condition	**True or False**	**State**
ACCEPT PHASE	True	When the Form Manager is performing accept phase processing.
CONVERTED	True	When the contents of a panel field have been converted to the data type of the form data item that corresponds to the panel field.
EMPTY FIELD	True	When the panel field that corresponds to the current activation item is empty (when a numeric field contains only zeros or a nonnumeric field contains only spaces).
	False	For text fields, sliders, waits, and icons.
FIRST DISPLAYED HORIZONTAL[1]	True	If the active item has the smallest *horizontal* subscript of all currently displayed occurrences of that item.
FIRST DISPLAYED VERTICAL[1]	True	If the active item has the smallest *vertical* subscript of all currently displayed occurrences of that item.
FIRST ITEM	True	When the current activation item is the first item on the activation list.
FIRST OCCURRENCE HORIZONTAL	True	If the active item has the smallest *horizontal* subscript of all occurrences of that item on the activation list.
FIRST OCCURRENCE VERTICAL	True	If the active item has the smallest *vertical* subscript of all occurrences of that item on the activation list.

Elementary Condition	True or False	State
FULL FIELD	True	When each character position in the active input field has been filled. This means: ● Format 1 picture fields (alphanumeric)—no spaces (blank characters) ● Format 2 picture fields (fixed decimal)—no leading zeros before and no trailing zeros after the decimal point
	False	For: ● Format 3 picture fields (floating point) ● Format 4 picture fields (date and time) ● Text fields, sliders, icons, and waits The presence or absence of a sign or a currency symbol does not affect FULL FIELD.
GROUP FIRST ITEM	True	When: ● Accept phase is just beginning. ● The current activation item corresponds to an item in a different group than the previous item processed. ● The previous activation item corresponded to an entity that was not a member of a group. The current item must be a member of a group.
GROUP LAST ITEM	True	When: ● The current activation item corresponds to an item that is in a different group than the next item to be processed. ● The next activation item corresponds to an entity that is not a member of a group.

Elementary Condition	True or False	State
		The current item must be a member of a group.
GROUP OTHER ITEM	True	When GROUP FIRST ITEM and GROUP LAST ITEM are both false.
HELP ACTIVE	True	When the help activation list is being processed.
HELP MESSAGE AVAILABLE	True	When HELP MESSAGE EXISTS is true and the operator has not seen the help message.
HELP MESSAGE EXISTS	True	When: ● There is help message text for the current activation item. ● The current activation item is a field, or icon activation item, when there is message help for the corresponding panel field, or for any of the groups or panels that contain this corresponding item.
	False	If the current activation item is a wait activation item, unless the item is a wait on panel and there is a help message at the panel level.
HELP PANEL EXISTS	True	When there is a help panel referenced at any level of form definition (field, icon, group, panel, or layout) for the current activation item.
IMMEDIATE	True	After the Form Manager encounters a POSITION IMMEDIATE, EXIT HELP IMMEDIATE, or RETURN IMMEDIATE response step.
LAST ITEM	True	When the current activation item is the last item on the activation list.
LAST DISPLAYED HORIZONTAL [1]	True	If the active item has the largest horizontal subscript of all occurrences of the current item displayed.
LAST DISPLAYED VERTICAL [1]	True	If the active item has the largest vertical subscript of all occurrences of the current item displayed.

Elementary Condition	True or False	State
LAST OCCURRENCE HORIZONTAL	True	If the active item has the largest horizontal subscript of all occurrences of that item on the activation list.
LAST OCCURRENCE VERTICAL	True	If the active item has the largest vertical subscript of all occurrences of the item on the activation list.
LEFTMOST ITEM [1]	True	If there is no active item on the current panel to the geographic left of the currently active item.
LOWERMOST ITEM [1]	True	If there is no active item on the current panel geographically below the currently active item.
OTHER DISPLAYED HORIZONTAL [1]	True	If the active item has neither the smallest nor the largest horizontal subscript of all occurrences of the current item displayed.
OTHER DISPLAYED VERTICAL [1]	True	If the active item has neither the smallest nor the largest vertical subscript of all occurrences of the current item displayed.
OTHER ITEM	True	When the current activation item is neither the first nor the last activation item on the activation list.
OTHER OCCURRENCE HORIZONTAL	True	If the active item has neither the smallest nor the largest horizontal subscript of all occurrences of the item on the activation list.
OTHER OCCURRENCE VERTICAL	True	If the active item has neither the smallest nor the largest vertical subscript of all occurrences of the item on the activation list.
PANEL FIRST ITEM	True	When the current activation item either is the first item on the activation list or corresponds to an item that is on a different panel than the previous item processed.
PANEL LAST ITEM	True	When the current activation item either is the last item on the activation list or corresponds to an item that is on a different panel than the next item to be processed.

Elementary Condition	True or False	State
PANEL OTHER ITEM	True	When the current activation item corresponds to an item that is neither the first nor the last item of a contiguous set of panel items. A contiguous set of panel items is two or more panel items that correspond to two or more adjacent activation items; the second activation item immediately follows the first, the third activation item immediately follows the second, and so on.
RIGHTMOST ITEM [1]	True	If there is no active item on the current panel to the geographic right of the currently active item.
UPPERMOST ITEM [1]	True	If there is no active item on the current panel geographically above the currently active field.
VALIDATED	True	After a RETURN response step (without IMMEDIATE) has been executed and after all items on the activation list have been validated.
VALIDATION STARTED	True	After the Form Manager begins validating a field activation item. This elementary condition is never true when the Form Manager is processing a wait activation item.

[1]All geographical elementary conditions are evaluated as false when the condition is compared against a WAIT item such as WAIT on PANEL.

Appendix B. Receive Control Text Items

Receive control text items return status information from the Form Manager to your application program. If you pass a receive control text item in your DECforms request, the Form Manager stores it in the receive control text.

Table B.1 lists and explains the receive control text items.

Table B.1. Receive Control Text Items

Control Text Item	Description
EI001	An error occurred during a DATE, TIME, or ADT data type conversion. These data types can be converted only to the DATE, DATETIME, TIME, ADT, CHARACTER NULL TERMINATE, or CHARACTER VARYING data types.
EI002	The number of records specified does not match the number specified in the record list.
EI003	The Form Manager could not convert a value from one data type to another data type, or data truncation occurred during the conversion.
EI004	The session was ended abnormally due to a severe error in another request.
EI008	DECforms software cannot process the request. An application AST is blocking the request.
EI009	Invalid device handler.
EI010	A previous CANCEL request is in progress.
EI011	Cannot close trace file.
EI012	Device I/O error.
EI013	A previous DISABLE request is in progress.
EI014	Cannot convert operands into common data type.
EI015	Fatal internal error.
EI017	Illegal FIELDVALUE context. Reference to FIELDVALUE outside of ACCEPT phase.
EI018	Illegal use of print viewport.
EI019	Database consistency check failed.
EI020	Invalid subscript range.
EI021	Invalid record count value.
EI022	Invalid record message descriptor.
EI023	No device handler exists for such device.
EI025	Specified parent request does not exist.
EI027	Specified output file cannot be opened.

Control Text Item	Description
EI028	Cannot open trace file for output.
EI029	Escape routine parameter has overflowed.
EI030	Address of the procedural escape was not found.
EI031	Request was terminated due to a severe error in an escape routine.
EI035	Length of the string is too small.
EI036	Cannot write to the trace file.
EI040	Attempted synchronous request while another request was active.
EI041	Shareable image mismatch.
EI042	Requested operation is not yet implemented.
ES000	Request calls completed.
ES001	External request ended abnormally because the timeout value specified in either the form or an external request argument expired before the operator finished giving input.
ES002	In an ENABLE request, the Form Manager was unable to find the form.
ES003	In an ENABLE request, either the Form Manager was not able to find a form that fit the display device or no layout was defined to match the definition of the FORMS$LANGUAGE logical name or FORMS$K_LANGUAGE item code.
ES004	In an ENABLE request, the Form Manager could not establish a session with *session-id* specified in the argument.
ES005	In a SEND, RECEIVE, TRANSCEIVE, or DISABLE request, the Form Manager lost the terminal connection, causing the previously established session to become lost.
ES006	In a SEND or TRANSCEIVE request, *send-record-name* specified a record name unknown in the form.
ES007	In a RECEIVE or TRANSCEIVE request, the *receive-record-name* specified a record name unknown in the form.
ES008	In a SEND or TRANSCEIVE request, the length specified in *send-record-message* (OpenVMS API) or in *send-record* (portable API) did not match the length of the record in the form.
ES009	In a RECEIVE or TRANSCEIVE request, the length specified in *receive-record-message* (OpenVMS API) or in *receive-record* (portable API) did not match the length of the record in the form.

Control Text Item	Description
ES010	Attempted to disable a session while a procedural escape was outstanding for that session.
ES011	Value specified by *session-id* of a request specifies a session unknown to the Form Manager.
ES012	The request was terminated using a RETURN IMMEDIATE response step.
ES014	Record in *send-record-name*, sent by the application program to the form, contains invalid information.
ES015	The length specified in *receive-shadow-record* during a RECEIVE or TRANSCEIVE request did not match the length of the shadow record in the form.
ES016	In a SEND or TRANSCEIVE request, the length of *send-shadow-record* was not 1.
ES017	Request interrupted by arrival of a CANCEL request.
ES019	No active requests to cancel.
ES024	Subscript reference is less than the base.
ES025	Subscript reference is greater than the defined array dimension.