HP Data Entry and Forms Management System (VPLUS) Reference Manual

HP 3000 MPE/iX Computer Systems

Edition 6

HEWLETT® PACKARD

Manufacturing Part Number: 32209-90024
E0300

U.S.A. March, 2000

Notice

Restricted Rights Legend

Acknowledgments

UNIX is a registered trademark of The Open Group.

Hewlett-Packard Company
3000 Hanover Street
Palo Alto, CA 94304 U.S.A.

Contents

Figures

Figures

Tables

1 VPLUS Overview

This chapter provides overview information about using VPLUS and also provides information about using this manual.

Product Overview

VPLUS is a comprehensive software system that provides an interface between a terminal and any transaction processing program. The interface can support the formatting, editing, and validation of data as well as the display and collection of that data. VPLUS also includes a source data entry facility that provides a "front end" for batch transaction processing applications.

Interface Design

The interface between the user at a terminal and the application program is implemented as a form. VPLUS enables you to easily design the layout of the form and the data to be displayed and entered on the form. A variety of processing for the data can be specified to occur before and after it is displayed and entered.

Transaction Processing

VPLUS provides a set of intrinsics that enables you to control forms and data on a terminal from an application program. These intrinsics are available for programs written in any of the supported programming languages listed in appendix A.

Additionally, some languages (also listed in appendix A) provide a special interface with terminals and forms, as described in their respective reference manuals. With these languages, you do not call VPLUS intrinsics directly. Instead, you specify the appropriate statements for the special interface.

Source Data Entry

VPLUS provides a ready-to-use data entry program, called ENTRY, for collecting data without any programming effort. The data is collected in a file for later batch processing. Users can browse the entered data using ENTRY and modify the data while browsing it. If you need additional or different capabilities that ENTRY does not provide, you can write your own application that incorporates VPLUS intrinsics. Appendix A, "Sample Programs", includes sample source data entry programs.

VPLUS also provides a batch reformatting capability. You can create specifications to control how collected data is to be reformatted, and then run a program to actually reformat the data.

The collection and reformatting capability, either singly or combined, provide a method of adding interactive source data collection to existing batch applications.

Features

The primary features of the VPLUS system are:

- A forms design program (FORMSPEC) that allows quick and easy interactive forms design using menus.
- Batch mode management of forms files, through FORMSPEC, that allows a forms file to be updated, compiled, and listed in a background job.
- Advanced forms design, through FORMSPEC, that provides for editing, formatting, movement, and computation of data when the form is executed, using the user's native language for alphabetic information and the local customs for numeric and date information.
- A set of intrinsics that manage the interface between:

  — an application

  — the terminal

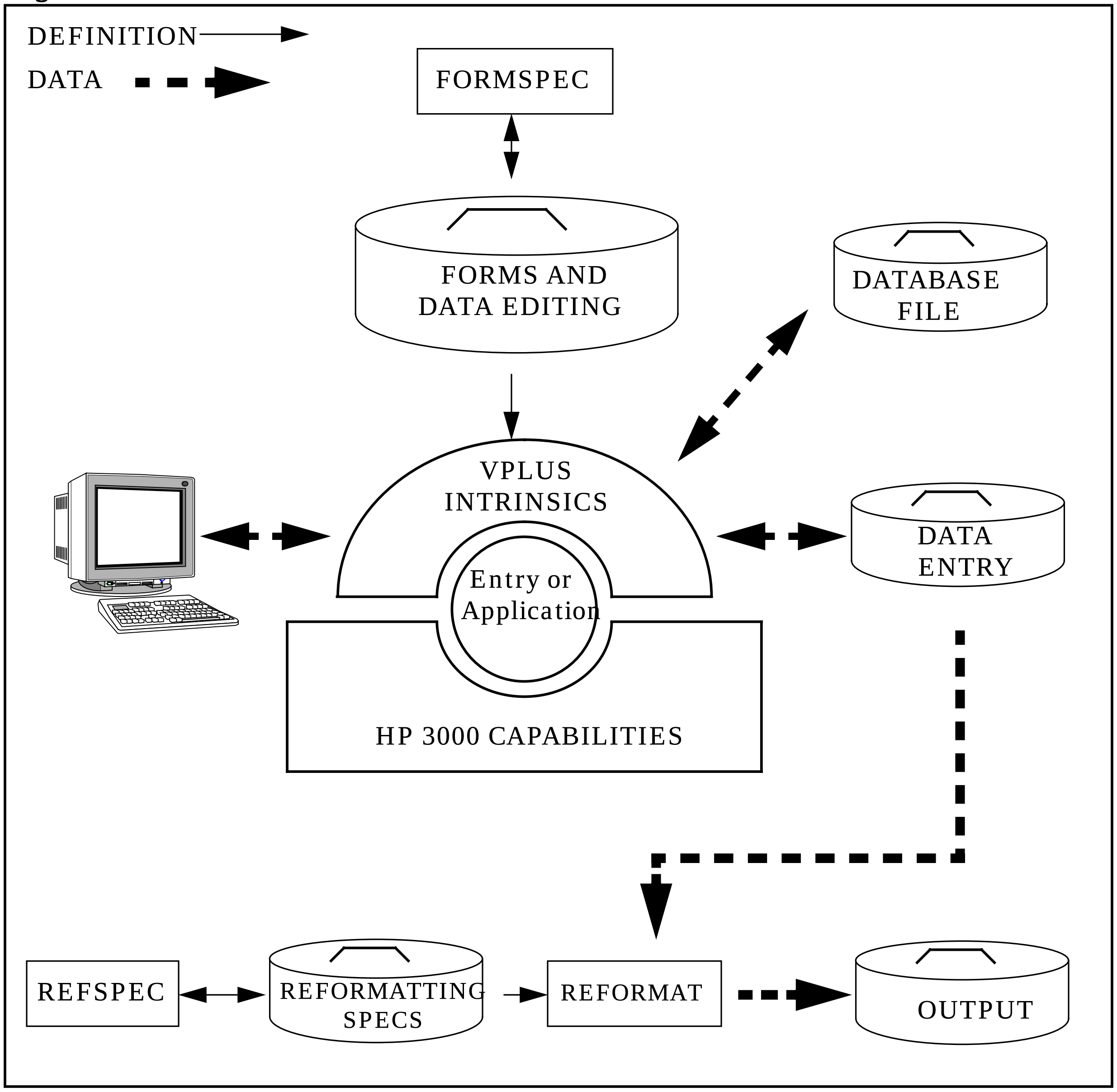
  — the FORMSPEC definitions

  — the data displayed and entered on the terminal

  These intrinsics manage the interface from applications written in any of the supported programming languages.
- A ready-to-run data entry program (ENTRY) that provides immediate use of forms for data entry and modification with no programming effort.
- A flexible data reformatting design program (REFSPEC) that specifies reformatting of data collected in a batch file.
- A batch program (REFORMAT) that reformats the collected data according to the REFSPEC formatting specifications and writes it to a file for use by an application.
- Run-time data transformation capability (Application-Ready Buffer, or ARB) that allows data to be entered on the screen in an order and format the user understands, while the application receives the data converted and reordered to match the format of its storage record.

Figure 1-1. shows an overview of the VPLUS system.

Figure 1-1. Overview of VPLUS



Designing Forms With FORMSPEC

FORMSPEC enables you to design forms ranging from simple to complex. FORMSPEC keeps forms in a special file, called a forms file. A forms file can have one or many forms. There may be multiple forms files per application or multiple applications per forms file, depending on the complexity of the task to be performed. You must compile the forms file before using it in an application program. You can modify forms at any time, but you must recompile the forms file to use the modified forms in your program.

You enter the form specifications on formatted screens, called menus, that FORMSPEC displays. The Main Menu and terminal function keys enable you to display all of the FORMSPEC menus. The various menus enable you to perform tasks, such as changing existing forms, adding new forms, and deleting forms or fields.

Levels of Forms Design

FORMSPEC has four complexity levels of forms design:

Simple Collection	You specify the form layout and FORMSPEC allows any character set codes that you enter. This level keeps the default field attributes and does not require any other FORMSPEC editing or formatting capabilities.
Simple Editing	You specify the form layout and specify edits for the field attributes, such as field type (optional, required, display only) or data type (character, numeric, or date). No processing specifications are required for the field attributes.
Full Field Editing	You specify field edits using processing specifications that apply to individual fields in a form. These include minimum length, range checks, pattern checks, etc. A subset of the available processing specifications are used for these edits.
Advanced Processing	You specify movement of data between fields and forms, arithmetic computation of data, formatting of data (`JUSTIFY, FILL, STRIP`), alteration of forms sequence, and conditional processing based on the result of processing specifications. This level implements the entire range of the processing specifications.

The following sections provide introductory information about each of these levels. Chapters 3 and 4 provide specific information on each level of forms design.

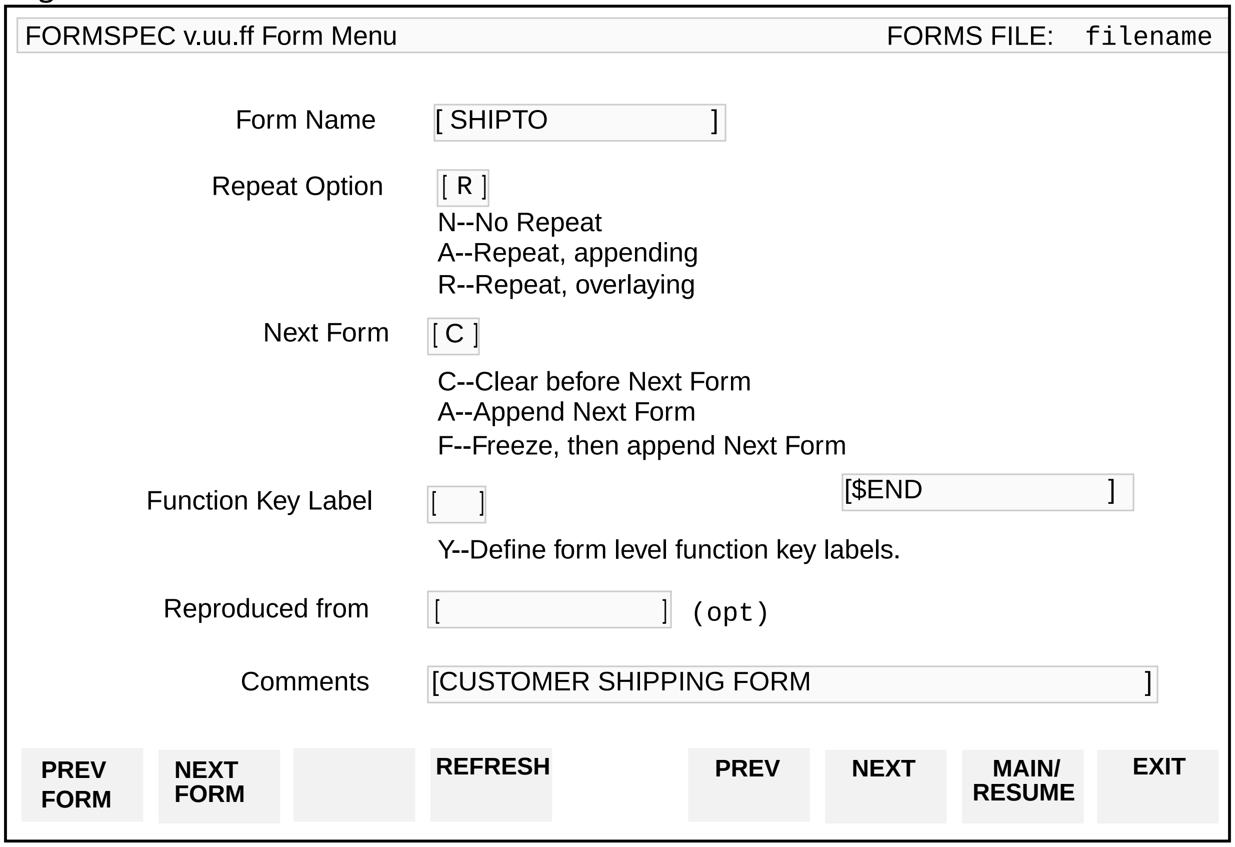
Simple Collection

When you run FORMSPEC, the first menu to appear is the Forms File menu in which you specify the name of the forms file. After you specify the forms file, the Main menu appears, enabling you to select one of the following tasks:

- Adding a form or a save field
- Selecting a terminal type or a native language
- Going to the global characteristics or to a particular form or field
- Listing a forms file or a form
- Deleting a form
- Copying a form
- Compiling the forms file
- Relating a child form to a parent form
- Going to the Application-Ready Buffer (ARB)

For example, if you want to select the task of adding a form, FORMSPEC would respond by displaying the Form menu, shown in Figure 1-2. You would use this menu to enter the form name and indicate how you want the form to be sequenced. This menu offers several options, such as appending to the current form. Chapters 3 and 4 instruct you about specifying whether the current form is to be repeated, or repeated and appended to itself.

Figure 1-2. FORMSPEC Form Menu

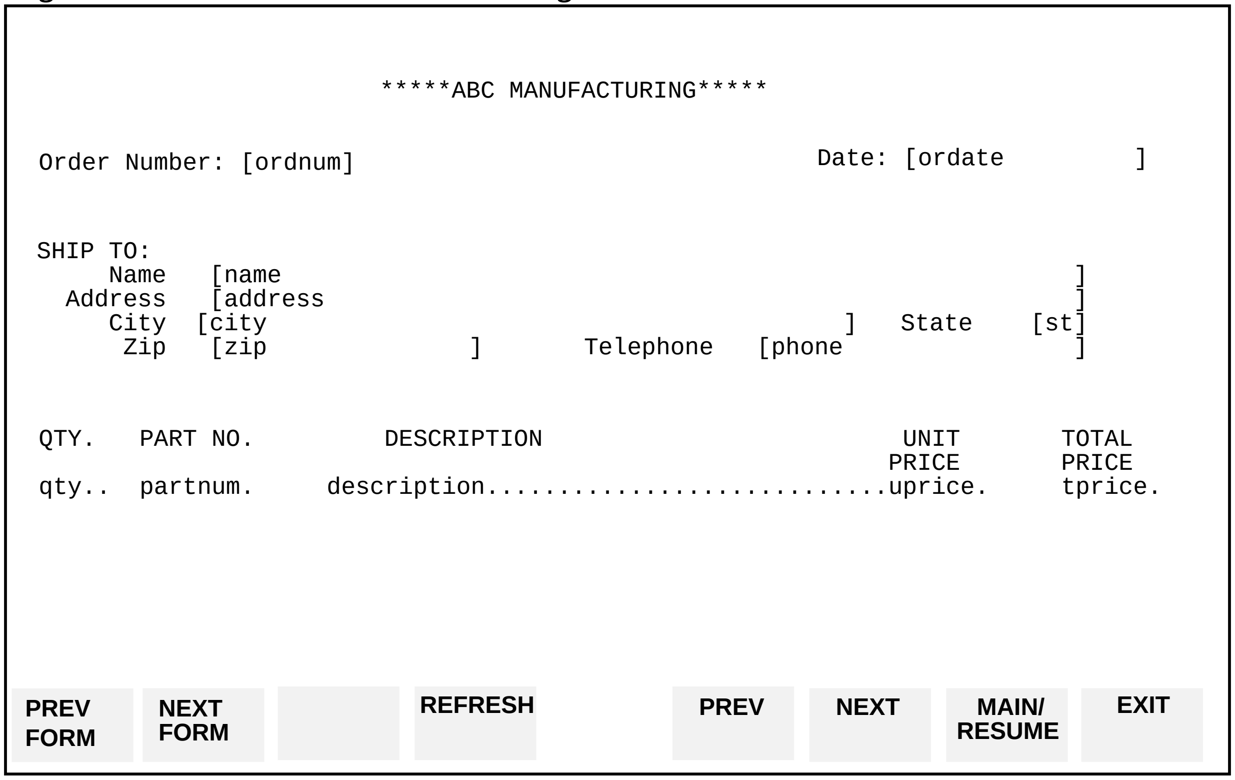


```
FORMSPEC v.uu.ff Form Menu                         FORMS FILE:  filename

          Form Name       [ SHIPTO          ]

      Repeat Option       [ R ]
                          N--No Repeat
                          A--Repeat, appending
                          R--Repeat, overlaying

          Next Form       [ C ]
                          C--Clear before Next Form
                          A--Append Next Form
                          F--Freeze, then append Next Form

 Function Key Label       [   ]                [$END             ]
                          Y--Define form level function key labels.

    Reproduced from       [             ]  (opt)

           Comments       [CUSTOMER SHIPPING FORM                  ]
```

After you make the desired selections, FORMSPEC provides you with a blank screen to design your form. You can use any of the terminal capabilities on this screen to insert or delete lines or characters, and to position the cursor. The special terminal capabilities that provide field enhancements, such as half bright, color, inverse video, or blinking are selected through FORMSPEC Field Menus, as described in chapters 3 and 4. You do not need to use complicated escape sequences to request enhancements to data fields.

The position of a field is defined by field delimiters. You identify each field by typing its name within these delimiters. If the field is too short for a meaningful name, you can provide a longer name later, if desired. The position of the beginning and ending of the field automatically defines the maximum length of each field. Field delimiters can be either brackets ([ ]) or invisible control characters, as shown in Figure 1-3. If the delimiters are not visible, you can use a fill character to make the actual length of the field visible during design.

Figure 1-3. FORMSPEC Screen Design



```
                         *****ABC MANUFACTURING*****

 Order Number: [ordnum]                              Date: [ordate        ]


 SHIP TO:
      Name   [name                                                    ]
   Address   [address                                                 ]
      City  [city                                        ]   State    [st]
       Zip   [zip                ]       Telephone   [phone           ]


 QTY.   PART NO.         DESCRIPTION                          UNIT       TOTAL
                                                             PRICE       PRICE
 qty..  partnum.     description.........................uprice.     tprice.
```

Simple Editing

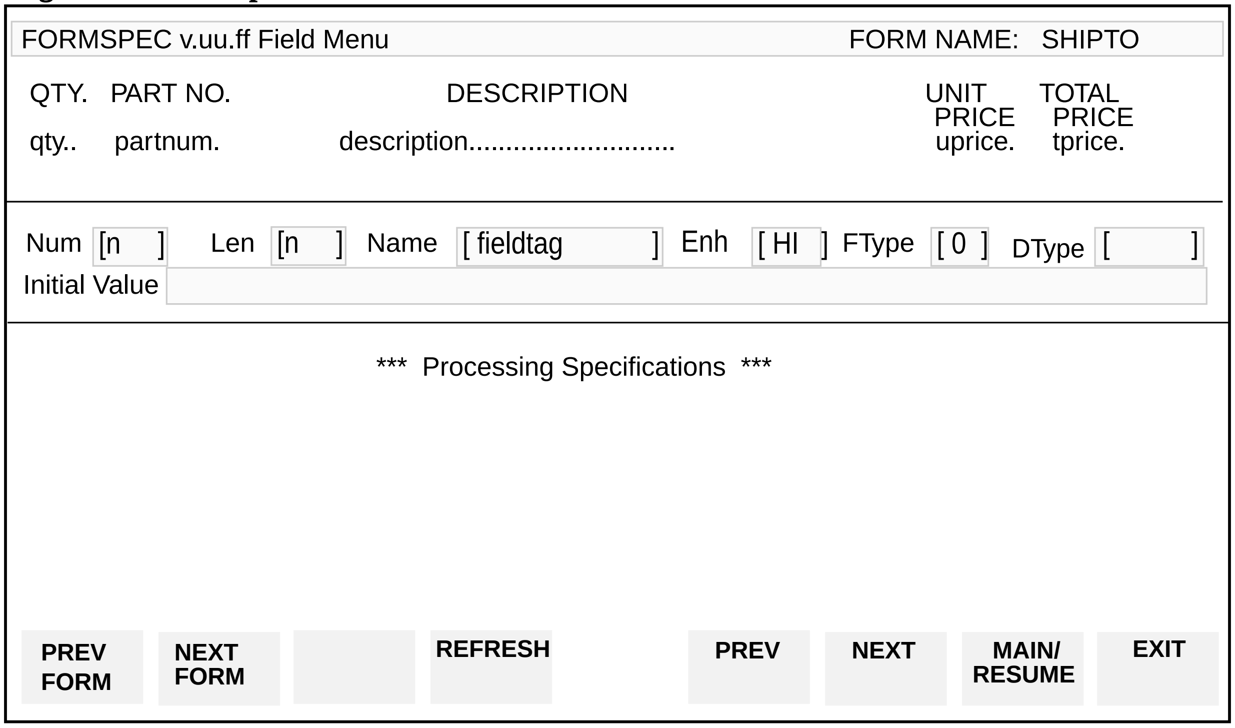
You can edit each field of a form by using the Field Menus. Figure 1-4. is a typical Field Menu example, which shows the various field attributes in the highlighted fields. This menu displays the field name and length determined by the form design. The menu also displays the data type, field enhancements, and the field type, any of which you can change by typing a new value over the displayed value. You can also assign the field a new name by which it is subsequently referenced, and you can specify an initial value for the field.

Field Name This parameter enables you to assign a field name longer than that assigned during form design. This is useful, because during form design, an identifier for each field typed within the field limits the field name to the length of the field.

Field Enhancement VPLUS displays each field in half bright, inverse video (code=HI) unless you change the value for the enhancement parameter on the Field Menu (or in the Globals Menu). Other enhancements, underline, blinking, color, and security can combine with or replace the default enhancements, or you can eliminate field enhancements altogether. (Only some terminals support color and security. Refer to "Supported Terminals Features" in appendix G for more information.)

Field Type This attribute controls how data is entered in the field. For example, if a field must contain a value, simply change the field type from the default value O (optional) to R (required) as shown in Figure 1-4. If a user then does not enter data in the field, an error message appears. If you want to prevent a user from entering data in the field because you plan to use it only to display data, change the field type to D (display only). You can force a field to be processed even if it is blank by changing the field type to P (process).

Figure 1-4. Sample Field Menu



```
FORMSPEC v.uu.ff Field Menu                              FORM NAME:   SHIPTO

QTY.   PART NO.              DESCRIPTION                         UNIT     TOTAL
                                                                  PRICE    PRICE
qty..   partnum.         description...........................   uprice.  tprice.

Num [n  ]    Len [n  ]  Name [ fieldtag     ]  Enh  [ HI ] FType [ 0 ]  DType [       ]
Initial Value [                                                                    ]

                       ***  Processing Specifications  ***
```

Data Type This attribute controls the type of data allowed in the field. Suppose you want to make sure that only digits are entered. Simply change the data type from the default type (CHAR for any character set code) to DIG. Or, if you want to allow entry of a signed number with two decimal positions and a decimal indicator, you can change the data type to NUM2 as shown in Figure 1-4. If the user enters a non-numeric value, an error message appears.

Initial Value You can use the Field Menu to specify a particular value to be displayed in that field when the form is first displayed at the terminal.

Full Field Editing

To specify full-field edits, you enter simple language statements in the lower portion of the Field Menu, labeled Processing Specifications. These field processing specifications enable you to check a field value for:

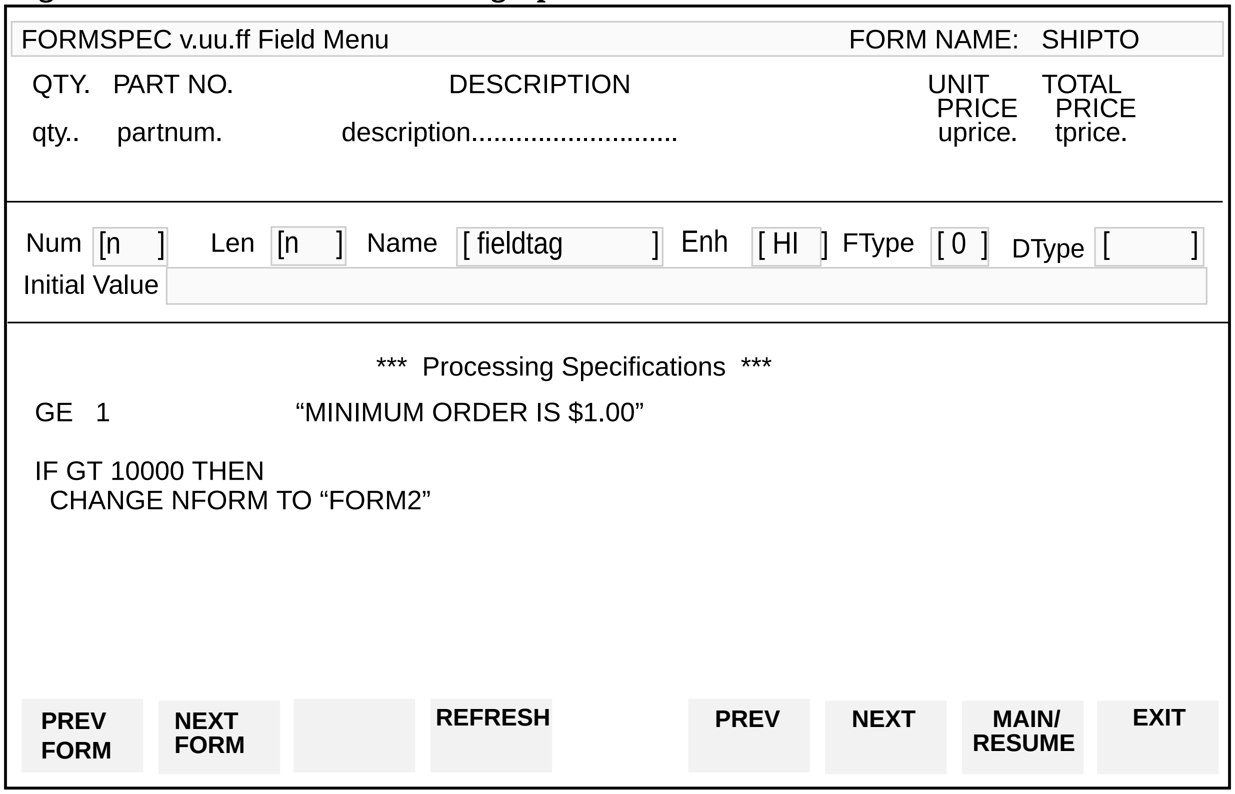
Minimum Length	The value entered must be at least a specified number of characters long. For example, `MINLEN 10` means at least ten characters must be entered in the field.
Equality	The value entered must be less than, less than or equal to, equal to, greater than or equal to, greater than, or not equal to a specified value. For example, `GE FIELD2` means the value entered must be greater than or equal to the value entered in a different field, `FIELD2`.
Range Check	The value entered must be within (or not within) a range of values. For example, `IN 10:20,F1:F4,100:300` means the value must be between 10 and 20, or between the

	values entered in fields F1 and F4, or between 100 and 300, inclusive.
Table Check	The value entered must be in (or not in) a table of values. For example, `IN 5,10,15,F7+5` means the value must be 5, 10, 15, or the current value of field F7 plus 5.
Pattern Check	The value entered must match a particular pattern. For example, `MATCH Aaa-ddd` means the value must start with the letter "A" and be followed by two alphabetic letters, a hyphen, and three digits.
Check Digit	A check digit in the entered value is checked according to modulus 10 or 11. For example, `CDIGIT 10` checks the value according to a modulus 10 check digit test.

Advanced Processing

Processing specification statements are similar to elements of a programming language in that the order of entry is significant. Figure 1-5. shows a Field Menu that includes advanced processing specifications.

Figure 1-5. Advanced Processing Specifications



```
FORMSPEC v.uu.ff Field Menu                          FORM NAME:  SHIPTO
 QTY.  PART NO.            DESCRIPTION                     UNIT     TOTAL
                                                           PRICE    PRICE
 qty..  partnum.       description...........................  uprice.  tprice.

 Num [n  ]   Len [n  ]  Name [ fieldtag      ]  Enh [ HI ] FType [ 0 ]  DType [      ]
 Initial Value

                        ***  Processing Specifications  ***
  GE  1                    "MINIMUM ORDER IS $1.00"

  IF GT 10000 THEN
    CHANGE NFORM TO "FORM2"
```

In this example, the `UPRICE` field (already specified as required and numeric) is limited further. The minimum value that may be entered in this field is 1. A custom error message `"MINIMUM ORDER IS $1.00"` appears if the user enters any value less than 1. Also, if the user enters a value greater than 10,000, a special next form, FORM2, appears when the user presses **Enter**.

Advanced Processing Statements The advanced processing statements provide:

Data Movement	You can set any field to a particular value, or to a value moved from another field. Default formatting of the data is performed during movement according to the data type of the destination field. For example, `SET TO !JUNE 17,1999!` moves the date constant (which must be delimited by exclamation points) to the current field. Another example, `SET F1 TO F3`, moves the value in the field F3 to the field F1.
Arithmetic Calculation	You can set any numeric field to a value calculated using standard arithmetic operators (+, -, *, /, %). For example, `SET TAX TO TOTAL*TAX_RATE` multiplies the value in the field `TOTAL` by the value in the field `TAX_RATE` and moves the result to the field `TAX`. (All these fields must be numeric.)
Data Formatting	You can specify particular formatting of the entered data in addition to default formatting. Formatting includes: `STRIP`, `JUSTIFY`, `FILL` and `UPSHIFT`. For example, `JUSTIFY RIGHT; FILL LEADING "0"` moves the data to the right of the field, and then fills any leading blanks with zeros.
Forms Sequencing	You can change the form sequence originally specified for the form on the Form Menu. For example, `CHANGE NFORM TO $HEAD` changes the next form to the head form. The head form is the first form displayed. One form is designated as the head form in the global characteristics. Another example, `CHANGE CFORM TO NO REPEAT`, stops the current form from repeating.
Conditionals	You can execute a processing specification, or group of specifications, but only if a particular edit is true. For example, `IF F1 EQ 20 CHANGE NFORM TO CLEAR` clears the current form and displays the next form when the value of field F1 equals 20.
Phases	You can execute processing statements in one of four phases: configuration (`CONFIG`), initialization (`INIT`), field editing (`FIELD`), and finish (`FINISH`).

Save Fields and Other Global Specifications

You can use the Save Field Menu to define special fields, called *save fields*, for the entire forms file. These fields are global to the forms file, and can be referenced in the processing specifications of any field in any form. Save fields are primarily useful for passing values between forms.

FORMSPEC supplies certain global characteristics of a form as defaults, such as field or error enhancements and the placement of the error/status line on the form. You can change these default characteristics using the Globals Menu.

Application-Ready Buffer

An application-ready buffer (ARB) is a buffer passed between the application and the run-time intrinsics for data collection and display. It performs the following functions:

- Holds data as the application will use it, which may differ from how it appears on the screen.
- Synchronizes with other application sources, such as the program code and databases.
- Facilitates maintenance in that the user can rearrange fields on the screen without modifying the application program. Conversely, changes in the application program that affect the ARB do not necessarily require changes in the screen.
- Supports an extended range of data types, including floating-point decimal and COBOL-packed and zoned decimal.
- Simplifies coding, because the field-by-field transformation is specified in the forms file rather than by using a series of transformation intrinsics in the application program.

The ARB presents data as the application expects to use it, such as to update a database, whereas the screen receives and displays data in the format that the user understands. This distinction between screen data and application-ready data facilitates maintenance.

Refer to Chapter 3 for information about creating an ARB.

Program Interface Intrinsics

All terminal-oriented applications can use a library of high-level intrinsics. Any of the supported programming languages listed in appendix A can call these intrinsics. Appendix A also contains listings of sample applications that use the intrinsics.

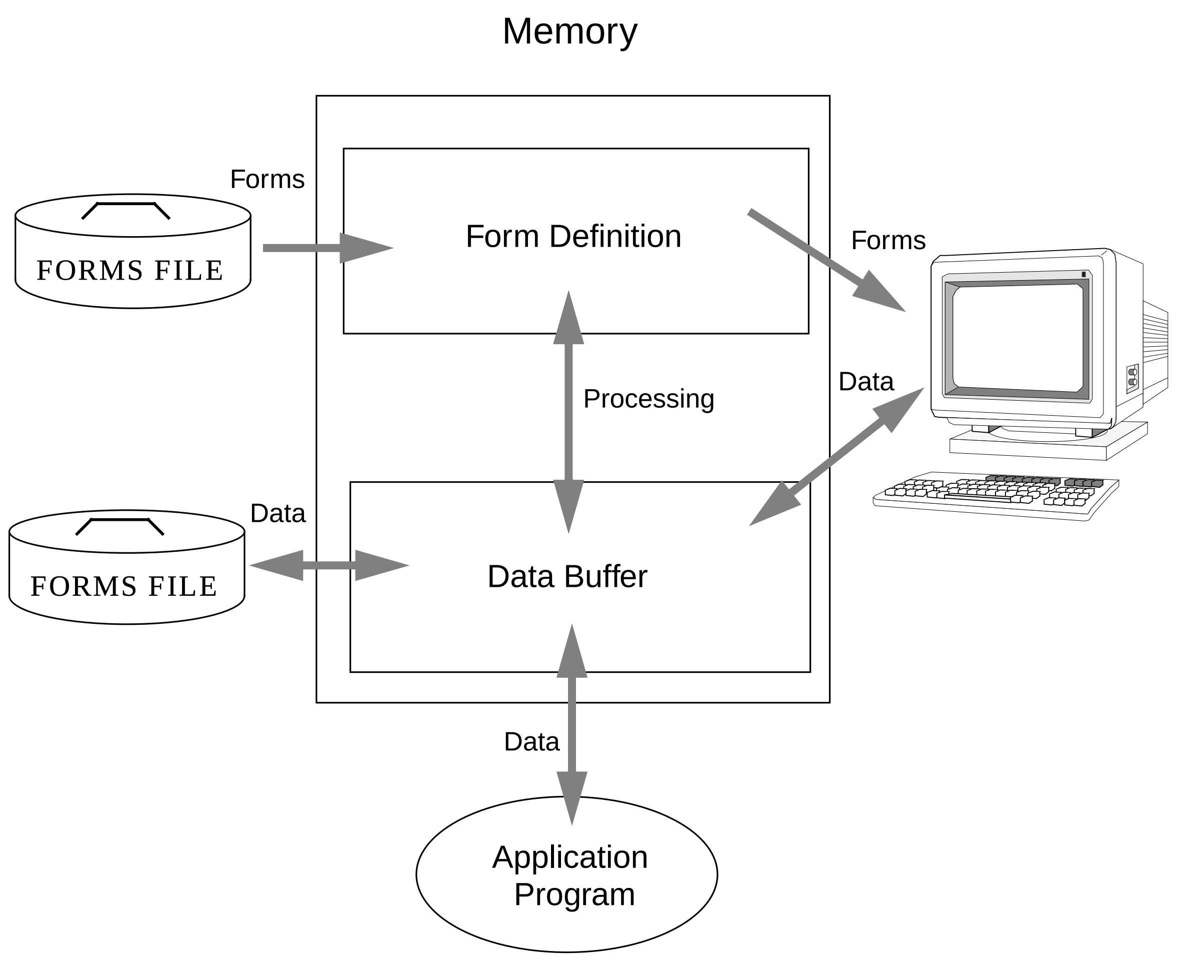
The program interface provides intrinsics for the following:

Terminal Interface	These intrinsics open and close a terminal file, load and unload forms (if the terminal has local forms storage), display a form and data to the terminal, and read data entered in fields of the displayed form.
Forms File Interface	These intrinsics open and close a forms file, get the next form in sequence from the forms file, and print the current form, with its contents, on the line printer.
Data Manipulation	These intrinsics initialize a form to its initial values, perform any user-defined edits, and perform any final form processing.
Access to Data	These intrinsics move data between the application program and VPLUS work areas. The data can be moved one field at a time or as a single buffer for the entire form. Data can be moved as character strings, or it can be transformed to and from internal representations most

	useful to the application program. The data transformations may be predefined for the entire buffer in the forms file using FORMSPEC (the ARB). If field-by-field movement is used, the transformation type is specified by the intrinsic name.
Data Transformation	These intrinsics gather data entered by the user at the screen and transform it into the format most useful to the application, or vice versa. Data passed to or from fields can be converted to or from a variety of data types.
Status/Error Control	These intrinsics set or capture error flags and display messages.
Batch Data Collection	These intrinsics open and close the batch file, write data to the batch file, or read data from the batch file.

Figure 1-6. shows an overview of the various transfers between elements controlled by the VPLUS intrinsics. Note that many applications use their own files or data bases rather than the VPLUS batch file for the collected data.

Figure 1-6. Transfers Controlled by VPLUS Intrinsics



Entering Data with ENTRY

VPLUS features a standalone data entry program, called ENTRY. Forms and data specifications are created using FORMSPEC. This program does the following:

- Displays forms at the terminal
- Accepts and validates data entered on the forms
- Writes the data to a batch file

ENTRY operates in two modes: data collection and browse/modify. Data collection mode collects data from the terminal, and browse/modify enables you to look at the collected data and modify it if necessary. When the user first runs ENTRY, the initial mode is always data collection. The user must request browse/modify by pressing a terminal function key.

Chapter 2 provides complete information about using ENTRY. The following sections discuss principles of data collection and browse/modify.

Data Collection

ENTRY displays forms on the terminal screen in the order that FORMSPEC determines. Each form is displayed with any initial values specified for the form. As each form is displayed, the user types data into the "unprotected" fields on the form. (These are the fields that permit data entry; they include all fields defined on a form except "display only" fields.)

After typing in the data, the user presses **Enter**. ENTRY then tests the entered data for errors and, if errors are found, it enhances each field in error. The particular enhancement is determined by the forms designer. The default error enhancement is inverse video underline. ENTRY also displays a message associated with the first field with an error. The user can then correct the error (or errors) and press **Enter** again. ENTRY continues to check the entered data until no errors are detected. It then writes the data entered on the form as a single record to the batch file, and displays the next form in form sequence.

Note that the forms designer determines the form sequence. However, the terminal function keys provide the user some control over this sequence. These keys enable the user to:

- Request the first (or head) form in execution sequence.
- Terminate a repeating form and display the next different form.
- Reset the current form to its initial values.
- Print the current form with its initial values on the line printer.
- Request browse/modify mode to view and/or change data already written to the batch file.
- Terminate ENTRY.

Browse/Modify

The user can view the data already written to the batch file by requesting browse/modify mode. The user requests this mode by pressing a terminal function key. The same key returns the user to data collection mode at the point of interruption.

When browse/modify is requested, the previous data record written to the batch file is displayed on the form through which it was entered. The user can examine the data, change it if desired, and then press **Enter**. Any new data is written to the same batch record, overwriting data previously entered in the same form.

If the user wants to examine the entire file, he or she simply presses a function key to request the first batch record. The data in this record is displayed on the form through which it was entered. When **NEXT** is pressed, the data in the next batch record is displayed. The user can progress through all the records in the batch file by making changes as required and pressing **Enter**, or by leaving the data unaltered and pressing **NEXT**.

As with data collection, terminal function keys enable the user to have further control over the sequence of browsed data. The keys for browse/modify allow the user to:

- Display data from the first batch record on its form.
- Display data from the previous batch record on its form.
- Display data from the next batch record on its form.
- Reset the current form to the values displayed before any current modifications were entered.
- Delete the record currently displayed at the terminal.
- Print the current record on the line printer.
- Return to collection mode to enter new data.
- Terminate ENTRY.

Reformatting Data

ENTRY writes entered data to a batch file. This file can then be used as input to an application. Sometimes it is necessary to reformat the data in the batch file so that it meets the input requirements of the application. VPLUS provides a reformatting capability that enables you to:

- Combine data entered on several forms into one output record.
- Separate data entered on a single form into several output records.
- Rearrange data within a record, inserting constants, and generating check digits before writing it to the output file.
- Format data within fields by justifying, filling, stripping characters, or adjusting the sign of a numeric value.

The REFSPEC program enables you to specify how you want to reformat the batch file. This program operates very much like FORMSPEC in that it displays menus, allows you to "draw" a pattern of the output record, and allows you to accept default field formatting or specify your particular formatting.

When you have created all of the reformat specifications, you compile the reformat file. The REFORMAT program reads data from the batch file, reformats it according to the REFORMAT file specification, and writes it to an output file.

REFORMAT can be run any time after a data entry batch file has been written. It can be run from a terminal or as a batch job, requiring only the names of the batch file, the reformat file, and the output file. The application can then use the output file it creates.

Refer to Figure 1-1., "Overview of VPLUS," for the flow of data between the batch file, through the reformatting specifications entered with REFSPEC, to the output file.

Using This Manual

This manual is intended for the following users:

Forms Designer	Designs the forms to be displayed, determines the order in which forms are displayed, and specifies any editing or special processing to be performed on data entered or displayed through the forms.
Applications Programmer	Designs and codes the application that uses data entered or displayed through the FORMSPEC forms.
Application User	Enters data on the forms defined with FORMSPEC for processing by an application or by ENTRY.

The following sections explain which chapters in the manual are applicable to each type of user.

Forms Designer

The forms designer who uses FORMSPEC to design forms should read chapters 3 and 4. These chapters describe basic and advanced forms design with FORMSPEC. The forms file can be managed in a job stream for many tasks, such as compiling and listing the file or renumbering fields within a form.

The forms designer with the responsibility for reformatting the data entered through the forms should read chapter 5, which describes how to specify reformatting with REFSPEC.

If designing forms in a native language other than the default (NATIVE-3000), the forms designer should refer to chapter 8 for information on Native Language Support.

The forms designer can skip chapters 2, 6, and 7, unless he or she is also the applications programmer.

Applications Programmer

The applications programmer should read chapter 6, which describes the VPLUS intrinsics callable from any of the supported programming languages listed in appendix A. If using one of the special interface programming languages, the applications programmer should refer to the reference manual for the particular language.

Since FORMSPEC provides editing and processing capabilities, the applications programmer should study FORMSPEC to determine how to divide the processing responsibilities between FORMSPEC and the application. Interactive FORMSPEC is described in chapters 3 and 4; batch mode FORMSPEC is described in chapter 7.

The applications programmer may also read chapter 5, which describes the specifications for reformatting data entered through an application like ENTRY. Reformatting may provide a way to adapt entered data to an existing application.

VPLUS Native Language Support enables an application programmer to create interactive applications that reflect native language numeric and date conventions for more than a dozen supported languages. For information on using VPLUS with Native

Language Support, the applications programmer should refer to chapter 8.

If the application being designed consists of data collection for later batch processing, the applications programmer should determine whether ENTRY fulfills the user's application needs. Chapter 2 describes ENTRY from the user's perspective. If the user's needs require application modifications, the applications programmer should review appendix A, "Sample Programs." This appendix lists the source code for ENTRY (in SPL) along with similar examples in various languages.

Application User

The application user is the person who actually enters data into the forms designed with FORMSPEC.

If the applications programmer has chosen to use ENTRY, read chapter 2 as an instruction guide for users or for their trainer. The chapter describes how to enter data to be processed by ENTRY.

If ENTRY has been modified, the applications programmer should modify chapter 2 to reflect the changes.

For other applications, the programmer should provide a similar guide that explains how to:

- Log on and off the HP 3000.
- Use the terminal keys, especially the function keys.
- Enter and modify data.

2 Entering Data with ENTRY

This chapter describes using the VPLUS standalone data entry application, called ENTRY. The forms file you design implements the specific data collection function that your application requires. ENTRY is the execution mechanism.

This chapter is intended for the operator of the ENTRY program. Since ENTRY is written using the VPLUS intrinsics, you can also use this chapter as a tutorial that explains some of the concepts you might apply when designing your own application program. If a locally modified version of ENTRY is used, the instructions in this chapter may also require modification.

Refer to chapters 3 and 4 for a description of forms design and to chapter 6 for a description of the intrinsics associated with ENTRY.

Protected and Unprotected Fields

Each form contains protected fields that you cannot change, and unprotected fields where you key in data. Protected fields can consist of field titles, report headings, or *display only* fields to which the system sends data. Data entered at a keyboard can only be keyed into unprotected fields.

Error Detection

If ENTRY detects an error after you enter data, ENTRY highlights the field(s) in error and displays an error message. You can then correct the error and reenter the data on the form.

Data Modification

After data has been entered, you can review it and modify it if desired. The entered data can be reviewed in a different order than it was entered.

Data Reformatting

You run REFORMAT if the file of collected data is to be reformatted before it becomes input to another program. The data is reformatted automatically and written to a selected output file from a selected batch file containing the previously entered data.

Terminal Usage

You can enter data using any of the terminals listed in appendix G. Refer to the appropriate terminal reference manual for complete instructions on terminal use.

Function Keys

ENTRY assigns special functions to the terminal function keys during data entry and subsequent data modification. These keys have slightly different functions depending on whether you are entering new data into the system, or are reviewing or modifying existing data. The keys are illustrated in this table, followed by their function for each mode.

Table 2-1. Function Keys for Entry

Key	Key Action Data Collection Mode	Key	Key Action Browse/Modify Mode
HEAD FORM **f1**	Display first form in sequence of forms.	FIRST REC **f1**	Display first record in batch file on form used to enter data.
f2	Not used; an error message appears if pressed accidentally.	DELETE REC **f2**	Delete current batch record from the batch file. Note that you cannot insert a record in place of a deleted record; any new records are added to the end of the batch file.
PRINT **f3**	Print current form on line printer. Prints form with current data (all values typed in fields and recorded by pressing **Enter**).	PRINT **f3**	Same as in Data Collection.

Table 2-1. Function Keys for Entry

Key	Key Action Data Collection Mode	Key	Key Action Browse/Modify Mode
REFRESH **f4**	Clear screen, initialize terminal, and redisplay with initial values. On terminals with local form storage, clear form storage memory and reinitialize form storage directory. Also used to recover from "Unexpected Program Interruption," as described later in this chapter.	REFRESH **f4**	Same as in Data Collection, except that previously entered data is displayed.
f5	Not used; an error message appears if pressed accidentally.	PREV REC **f5**	Display previous record in batch file on form used to enter data.
NEXT FORM **f6**	Interrupt display of repeating form, and display next form.	NEXT REC **f6**	Display next record in batch file on form used to enter data.
BROWSE **f7**	Enter browse/modify mode. The previous form is displayed with entered data; the window line is updated to show BROWSE.	COLLECT **f7**	Return to Data Collection mode; the window line is updated to show COLLECT.
EXIT **f8**	Exit from ENTRY; return to MPE control.	EXIT **f8**	Same as in Data Collection.

Function key labels for a form can be defined with FORMSPEC as described in chapter 3. The labels you define with FORMSPEC are displayed in place of the ENTRY labels; however, changing a label also affects the action associated with the key.

Display Enhancements

The terminals used for data entry provide display enhancements to highlight portions of the forms display. These enhancements are defined during forms design and vary depending on how the form is defined. The enhancements may highlight both input and display-only fields. An alternate enhancement may be defined to indicate fields in which errors are detected. The data capture devices do not support display enhancements.

If not specifically changed by the forms designer, all unprotected fields are shown in half-bright inverse video. If a field contains an error, it is normally highlighted using full-bright inverse video, and is underlined. (Note that the particular way a field in error is enhanced depends on choices made by the forms designer. In general, enhancements enable you to easily distinguish fields where data can be entered and to locate fields where data has been entered incorrectly.

Depending on the design, fields also may be delimited by brackets to indicate the exact location of the field within the form.

Printing Forms and Data

To print any form, simply press **PRINT**. In Collect mode, the current form with any initial data is printed on the line printer. Unless a specific initial value has been assigned to a particular field by the forms designer, all fields in the form are blank initially. In Browse/Modify mode, the form is printed with the data previously collected to the current record.

Ordinarily, the printed form contains the data that appears on the screen. However, if you key in data on the form and press `PRINT` before you press **Enter**, the new data is not printed. In Collect mode, only initial data displayed with the form is printed. In Browse/Modify mode, any data already recorded on a batch record is printed.

Running ENTRY

To run ENTRY, log on to the HP 3000 and type the following command:

```
:RUN ENTRY.PUB.SYS
```

Then press **Return**.

Specifying ENTRY Files

After you start ENTRY, it prompts you for a forms file name where the data entry forms are stored. It also prompts for the name of the batch file where you want to save the data. If you do not know the names of these files, ask your application manager.

The forms file is a standard MPE file identified by its data file name. It may be fully qualified by account, group name, and lockword.

The batch file is a standard MPE file. If the named batch file does not exist, a new file is created automatically. If the file already exists, it is opened so that new data can be added to the end of the existing data in the file.

Responding To ENTRY Prompts

ENTRY prompts for the forms file name as follows:

```
ENTER FORMS FILE NAME AND PRESS RETURN:
```

Type the desired forms file name and then press **Return**. If a password is associated with a forms file, you are prompted for it twice. The forms file must be opened twice: first to determine the forms file type (KSAM or non-KSAM), and then to open it in the correct mode. If you want to avoid being prompted twice for the forms file password, enter it along with the forms file name in the format: *formsfile/lockword*.

ENTRY then prompts for the batch file name:

```
ENTER BATCH FILE NAME AND PRESS RETURN:
```

Enter the batch file name. If the batch file name you entered is an existing file to which data has already been written, you may receive the following message:

```
WARNING: Forms file recompiled since this
batch was created. Enter "Y" to continue:
```

This message is issued if the forms file has been modified and recompiled since it was last used to collect data to the batch file. Enter Y to continue only if you are sure that the changes to the forms file will not invalidate data already entered in the file. Otherwise, press **Return**. This causes the forms file prompt to be reissued so you can enter a different forms file name and/or a new batch file name. Generally, you should use a new batch file when a forms file is recompiled.

If the batch file you named was originally used with a different forms file than the one you named, you will receive this message:

```
WARNING: A different forms file was used to
create this batch. Enter "Y" to continue:
```

This message may mean that the wrong forms file was entered. If you press **Return**, the forms file prompt is issued again so you can enter the correct forms file name. If the forms file name was correct, but the batch file name is wrong, you must reenter the forms file name and then enter a different or new batch file name when the batch file prompt is issued. If this situation occurs, you would typically not enter Y to continue, because you would not use a batch file that does not match the forms file.

If the forms file has been designed to accept data using a native language other than NATIVE-3000 and ENTRY does not know the exact language that should be used, ENTRY displays this message:

```
ENTER LANGUAGE ID NUMBER AND PRESS RETURN:
```

If you see this message and do not know the appropriate number, contact your application manager. For more information on Native Language Support, refer to chapter 8.

Removing Deleted Records From a Batch File

When records are deleted from a batch file, they are only "tagged" for deletion. If you wish to physically remove deleted batch records, use the HP file copier, FCOPY, as follows:

1. Determine the record size of the batch file (in bytes) by using the MPE :LISTF command. Assign this value to N.
2. The delete flag is located 19 bytes from the end of a batch record, or at location N-19. Assign this value (N-19) to M.
3.
```
:RUN FCOPY.PUB.SYS
>FROM=oldbatch;TO=newbatch;SUBSET=#0,0#,M;NEW
>EXIT
```
4. If desired,
```
:PURGE oldbatch
:RENAME newbatch,oldbatch
```

Expanding the Batch File

In most cases the batch file that ENTRY builds for you has enough space, but if you want to enter a large amount of data (more than 1023 records), you must either specify a larger batch file before running ENTRY or increase the size of your existing batch file. To specify a larger batch file, first use the MPE :FILE command before running ENTRY:

```
:FILE filename;DISC=numrec
```

where *filename* is the name of a new batch file and *numrec* is the number of records you want the new batch file to have. When you run ENTRY, specify **filename* as the batch file; the program will build it with the number of records you requested.

You can also use FCOPY to enlarge an existing batch file. First issue the MPE :FILE command as shown above. Then run FCOPY to build a new, larger file of the size you requested and copy the existing data into it:

```
:RUN FCOPY.PUB.SYS
```

When the FCOPY prompt (>) appears, type:

```
>FROM=oldname;TO=*filename;NEW
```

where *oldname* is the name of your existing batch file and *filename* is the name of your new, larger file. Remember to include the asterisk; otherwise, the new file will not be larger than the old one. After the prompt reappears, type EXIT and press **Return**.

Finally, purge the old batch file from the system and assign its name to your new batch file:

```
:PURGE oldname
:RENAME filename,oldname
```

where *oldname* and *filename* are the names of your old and new files, respectively. When you run ENTRY again, specify *oldname* as the batch file, and you can continue entering data as before.

Block Mode

After you have specified the ENTRY files, ENTRY operates in block mode. When you log on and request ENTRY, and when you specify the forms and batch file names, your commands are entered into the system by pressing **Return** at the end of each line. When ENTRY operates in block mode, your data is entered only when you press **Enter**. This allows you to move around on the screen, pressing **Return** if you wish, and to type in or correct data. ENTRY does not receive any keyed data until you press **Enter**.

After you have specified the forms and batch files, ENTRY begins to display the forms defined in the specified forms file. The data you enter in each form displayed on the screen is stored in separate records of the specified batch file. Subsequently, you can look at or modify the data stored in the batch file as described in the section entitled "Modifying Data."

Local Form Storage

Some terminals have a feature that allows forms to be stored locally in terminal memory. Forms stored in the terminal can be displayed on the screen directly rather than having to be retrieved from the memory area of the computer. This feature, with look ahead mode, is automatically activated when you run ENTRY. Refer to appendix G for a list of supported terminals with this feature; refer to chapter 6 for more information on local form storage.

Forms Sequence

After you specify the forms and batch files, the *Head* form is the first form to appear. Depending on the forms file definition, the Head form could be a menu from which you could select the particular form on which you want to enter data. If the Head form is a menu, the data you enter determines which form is displayed next.

The next form displayed after the Head form also depends on the definition of the forms file; it is usually a data form on which you enter data in unprotected fields. All entered data is stored in the batch file when you press **Enter**. At that time, ENTRY automatically displays the next form in the sequence defined in the forms file. This process continues until all the forms in the sequence have been displayed, or until you press the Exit function key.

Repeating Forms

The next form is not always a different form. For instance, a form used for order entry may be repeated as you enter different data into the form. Each time you press **Enter** to enter data for a repeating form, the next form displayed is identical to the preceding form except that the unprotected fields are empty or contain initial values.

You interrupt a repeating form in order to display the next form by pressing the NEXT FORM function key. In this case, the form that immediately follows the repeating form is displayed. If you know that the next form is the first form (the "head" form), you can also display this form by pressing the HEAD FORM function key.

Appended Forms

Forms may also be designed so that the next form is appended to the current form. An appended form is displayed immediately below the current form on the screen. When you type data into the current form and then press **Enter**, the data in the current form is written to the batch file as usual, but the form and its data remain on the screen with the next form displayed below it. Note that an appended form may also be a repeating form, or it may be different from the preceding form.

You may then enter data in the appended form, but not in the previous form. Although the previous form remains on the screen, all of its fields become protected.

Frozen Forms

A form can be designed as *frozen*. A frozen form remains on the screen when subsequent forms are displayed. The next form after a frozen form is always appended to the frozen form. As you enter data into the next form (or forms), eventually no more space will be available on the screen, depending on the form size. At this point, the top appended form rolls off the screen leaving the frozen form at the top of the screen. Forms rolled off the screen cannot be rolled back down for review. Data in the frozen form can be changed only by entering Browse/Modify mode using the BROWSE function key.

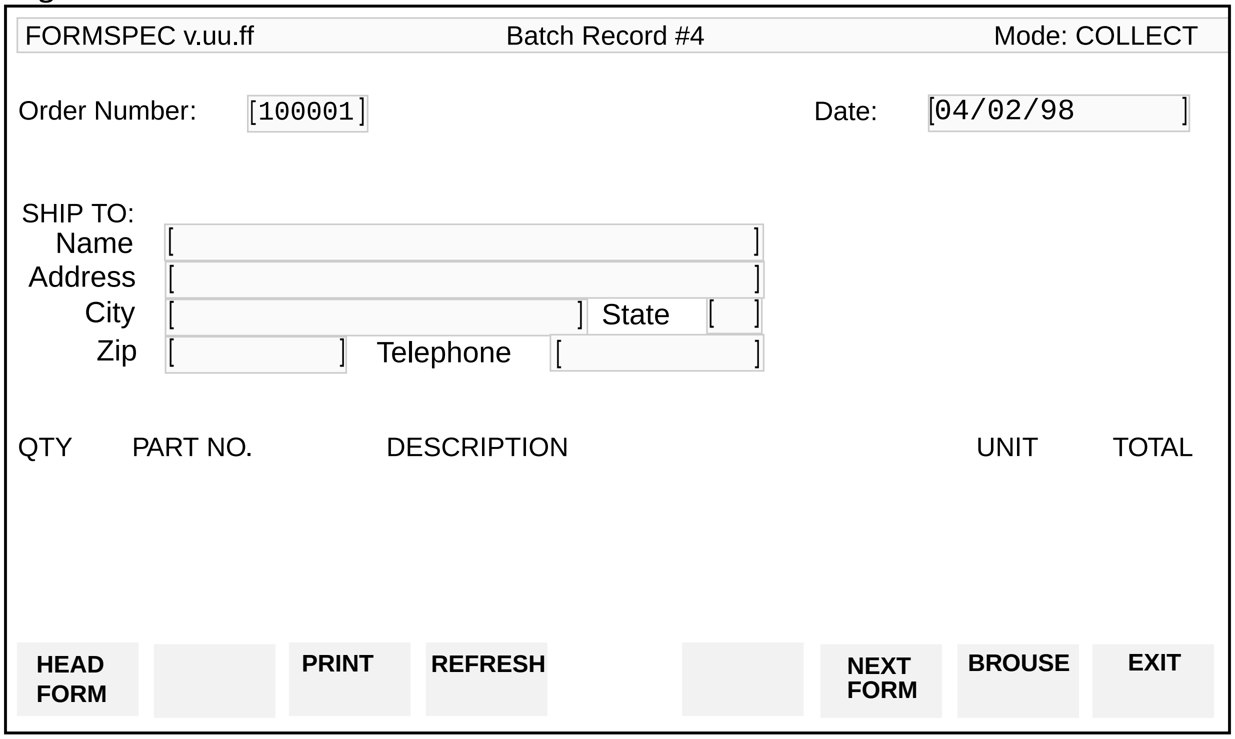
The following steps provide an example of a possible sequence of forms, including a repeating appended form that follows a frozen form.

1. For a frozen Head form, press **Enter** to append the Next form, as shown in figure 2-1. The Next form is then repeated/appended whenever you press **Enter**.
2. Press `NEXT FORM` to stop repeating, clear the screen, and display the Next form.
3. Press **Enter** to return to the Head form.

Entering Data

You can enter data only in *unprotected* fields. Depending on how the forms are designed, unprotected fields may be delimited by brackets as shown in Figure 2-1, they may be designated simply by the enhanced display as shown in Figure 2-1, or they may not be distinguished from the protected areas of the screen. Press **Tab** to position the cursor to the beginning of the next unprotected field.

Figure 2-1. Bracketed Fields



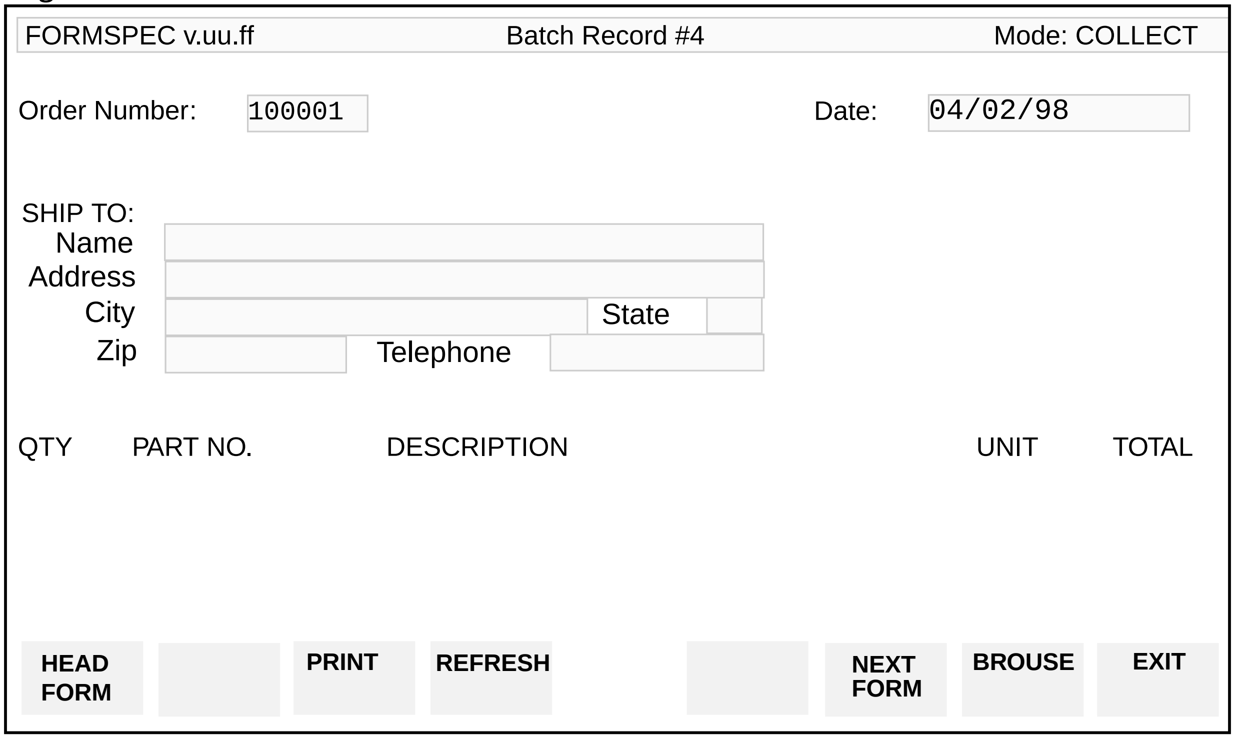
```
FORMSPEC v.uu.ff                Batch Record #4                    Mode: COLLECT

Order Number:    [100001]                          Date:    [04/02/98     ]

SHIP TO:
    Name    [                                ]
  Address   [                                ]
     City   [                       ] State  [  ]
      Zip   [         ]   Telephone  [          ]

QTY     PART NO.        DESCRIPTION                          UNIT      TOTAL
```

You enter data anywhere within the unprotected field. The brackets, although enhanced like the field, cannot be overwritten by data. If the data you key into a field does not fill the field, press **Tab** to go to the next field.

After keying all the data into the unprotected fields on the screen, press **Enter**. ENTRY collects the data and tests for errors. If no errors are detected, the data is then written as a record to the batch file you named when you first ran ENTRY.

Each time you press **Enter**, a new record is written to the batch file. Each batch file record is associated with the data entered on a single form. If a number of appended forms are displayed on the screen, the data you enter on each form is written to a separate record in the batch file. (The relationship of records to forms is important during the browse/modify phase of data entry, described in the section entitled "Viewing and Modifying Data".)

Figure 2-2. Fields without Brackets



FORMSPEC v.uu.ff Batch Record #4 Mode: COLLECT

Order Number: 100001 Date: 04/02/98

SHIP TO:
Name
Address
City State
Zip Telephone

QTY PART NO. DESCRIPTION UNIT TOTAL

Optional and Required Fields

Fields can be defined as required or optional. If a field is required and you do not key in data, ENTRY detects an error and displays an error message. If a field is optional, ENTRY accepts an empty field.

Program-Generated Data

When you press **Enter**, ENTRY can assign values to certain fields in the form if the appropriate processing specifications have been supplied during forms design. With the appropriate processing specifications, ENTRY can calculate values from data that you have entered. It can also move values from other fields in this form or another form or can specify actual values to be displayed in these fields.

For example, you might enter the quantity, the unit price, and the part number of an item. ENTRY uses these values to calculate the net price for this line of the order and, when you are in Browse mode, will display the price in a field on the form. Figure 2-6 illustrates this example. Another application for these calculations is that you could enter a value that ENTRY automatically edits. For example, a date you enter as `September 15, 1989` could be displayed in the same (or a different) field as `9/15/89`.

The following steps provide an example of assigning values to fields.

1. Key in values for `QTY, PART NO., DESCRIPTION` and `UNIT PRICE`.
2. Press **Enter**.
3. ENTRY calculates `NET PRICE` and displays it in Browse mode.

Correcting Errors

You can correct errors either before or after pressing **Enter**. If ENTRY detects an error after you press **Enter**, ENTRY does not write the data to the batch file until you correct all errors in the form. (Data can be changed after being written to the batch file with the Browse/Modify capability described in the section entitled "Modifying Data.")

Before Pressing Enter

You can examine the form for errors before pressing **Enter**. If you notice an error, you can correct it and then press **Enter**. Refer to your terminal reference manual for information on how to use the cursor positioning and editing keys.

All unprotected fields on the screen can be cleared to the original spaces or default values by pressing the REFRESH function key.

NOTE If this is an appended form, any previous forms on the screen will be lost. If forms can be stored locally in the terminal for fastest display, they will be flushed from the local storage area and normal display will occur.

After Pressing Enter

After you have pressed **Enter**, ENTRY edits the data you have entered. If ENTRY detects any errors, it leaves the form and the entered data on the screen, positions the cursor to the beginning of the first field with an error, and causes all fields with errors to blink (or, depending on the forms design, to be enhanced in another manner). Also, a message describing the first error is displayed in the window line. This line is dedicated to error and status messages, and it appears on the form in the position specified in the forms file with FORMSPEC.

You should correct the field in error and then press **Enter** again. If more than one field contains errors, you can correct all of them before pressing **Enter**.

System and Logic Errors

Some errors cannot be corrected as described above. When a system error occurs, the program terminates and returns to MPE control. An MPE error message is displayed on the screen. System errors are caused by problems in the computer system. Other errors are logic errors, which do not terminate the program. A logic error is not necessarily the fault of the user, but may result because of a form design flaw.

A logic error might occur, for example, when data entered causes the program to perform an impossible calculation, such as division by zero. Whether you encounter a system or a logic error, you should consult with your application manager for the best method of correcting the error.

Interrupting Data Entry

If you want to interrupt data entry before reaching the last form, press **Enter** to record the data on the current form, then press EXIT. The next time you run ENTRY with the same forms file and the same batch file, the next form is displayed automatically, enabling you to resume data entry from the point where you had stopped.

If you press EXIT before pressing **Enter**, data keyed into the current form is not recorded in the batch file. When you run ENTRY again with the same forms and batch files, the last form is redisplayed with any initial values, and you must retype the data into the form whose data entry you interrupted in the previous session.

Terminating a Session

You can terminate ENTRY by pressing EXIT to return to MPE control. The effect is the same as for interrupting data entry. After the MPE colon prompt (:) appears, you terminate the session by entering the BYE command and pressing **Return**, as follows:

```
:BYE
```

Unexpected Program Interruption

The program may be interrupted unexpectedly because of a power failure or because you accidentally pressed **Break**. Refer to appendix G for instructions on program recovery. Once you have recovered, the form displayed at the time of the failure reappears, and you can continue where you had stopped before the interruption.

Viewing and Modifying Data

You can view and modify data written to a batch file in data collection mode by using Browse/Modify mode.

Invoking Browse/Modify Mode

If you are currently in Data Collection mode, press BROWSE to invoke Browse/Modify mode. If you have exited from ENTRY, run the program again. When the forms file menu appears on the screen, enter the name of the forms file used to enter the data and the name of the batch file to which the data was collected. ENTRY displays the next form in which you can enter data. To enter Browse/Modify mode in order to view data already entered, press **BROWSE**.

When you press **BROWSE**, the data in the last record of the batch file to which data has been written is displayed. If you then press **PREV REC**, a previous record appears. You can also press **NEXT REC** to display the next record in a forward direction. Of course, you can only view data that has already been entered.

You can press **FIRST REC** to display the data from the first record in the batch file. Then you can press **NEXT REC** to proceed sequentially through the records in the batch file. You can continue to press **NEXT REC** until you have viewed each record in the file that contains data. When the last form with data has been displayed, pressing **NEXT REC** displays the following message:

```
There are no more batch records
```

If you press **PREV REC** after displaying the first form, ENTRY responds with:

```
There are no previous records
```

Resuming Data Collection

You can press **COLLECT** at any time to return to data collection mode. The next form in sequence appears. You can then continue to enter data into the batch file that you have been reviewing in Browse/Modify mode.

NOTE The **BROWSE** and **COLLECT** keys are physically the same f7 function key. In data collection mode, the key is called BROWSE; in Browse/Modify mode it is called **COLLECT**.

Viewing Data

When you browse the data in the batch file using **PREV REC** or **NEXT REC**, the selected form appears with data previously entered at the keyboard or generated by the system. That is, you see the form with calculated data as well as the entered data exactly as it appeared when you pressed **Enter**.

Only one record of data from the batch file appears on the screen with each selection of **PREV REC** or **NEXT REC**. If the data originates from a set of appended forms, the forms and data append when you press **NEXT REC**. However, when you move backward by pressing **PREV REC**, forms originally appended are displayed one at a time with data.

Modifying the Data

You can modify any unprotected field by typing over the data displayed in the field. When you press **Enter**, the modified data is edited and written to the associated record in the batch file. Existing values are overwritten by the new values.

When you change a field, ENTRY determines if the change affects other fields in the form. For example, if you change the quantity of an item on an order form, ENTRY recalculates the net price if, during normal data entry, ENTRY had calculated net price from the quantity and a unit price.

ENTRY rechecks the modified data according to any specified editing. If you change one field which consequently causes another field dependent on it to fail the edit check, you must correct the field that is now in error.

You can use **REFRESH** to reset the data on a form during modification. However, in Browse/Modify mode, ENTRY redisplays the data previously collected to the batch file, replacing any changes just entered; it does not clear the fields to blanks or reset initial values.

Figures 2-8, 2-9, and 2-10 show an example of data modification.

1. Press **BROWSE** to enter Browse Mode and display the previous batch record. Since this is an appended form, you would only see the last appended form with data.
2. Press **PREV REC** until the record to be modified, Batch Record #1, appears. Modify TAX RATE to 0.04 and press **Enter**. ENTRY recalculates SALES TAX and TOTAL PRICE and displays the next form.
3. Press **PREV REC** again to see the recalculated values.
4. Press **COLLECT** to continue data collection.

Deleting Data

You can delete an entire record from the batch file by pressing **DELETE REC**. Thereafter, ENTRY skips over this record when you browse through the data in the batch file.

You cannot insert a new record in the same position as the deleted record. ENTRY does not provide for inserting records; any new records are written to the end of the batch file.

If you only want to delete a single field, modify the field to show all blanks and press **Enter**.

3 INTRODUCTION TO FORMS DESIGN

The forms displayed to a user are designed at a terminal with the interactive program, FORMSPEC. Using FORMSPEC, the designer designs the forms and defines parameters for editing, manipulating and displaying data. Forms defined in this manner are saved in a forms file. At any time during forms design, FORMSPEC allows the designer to modify the forms in the form file. Once a form has been defined, the designer can create an Application-Ready Buffer (ARB) for it. The ARB stores screen data in a format compatible with the requirements of the application program: this format may differ from the screen layout.

GETTING STARTED WITH FORMSPEC

You execute FORMSPEC in interactive mode by entering the following command in response to the MPE colon prompt:

```
:FORMSPEC.PUB.SYS
```

FORMSPEC runs entirely in block mode. HP block mode terminals are placed in block mode automatically by VPLUS; refer to Appendix G for a list of supported terminals.

When you run FORMSPEC, it displays a series of "menus" on which you enter the specifications to define the forms for a single application. Each menu uses a fill-in-the-blanks type of format. Refer to the menu descriptions later in this section for more information on all the FORMSPEC menus.

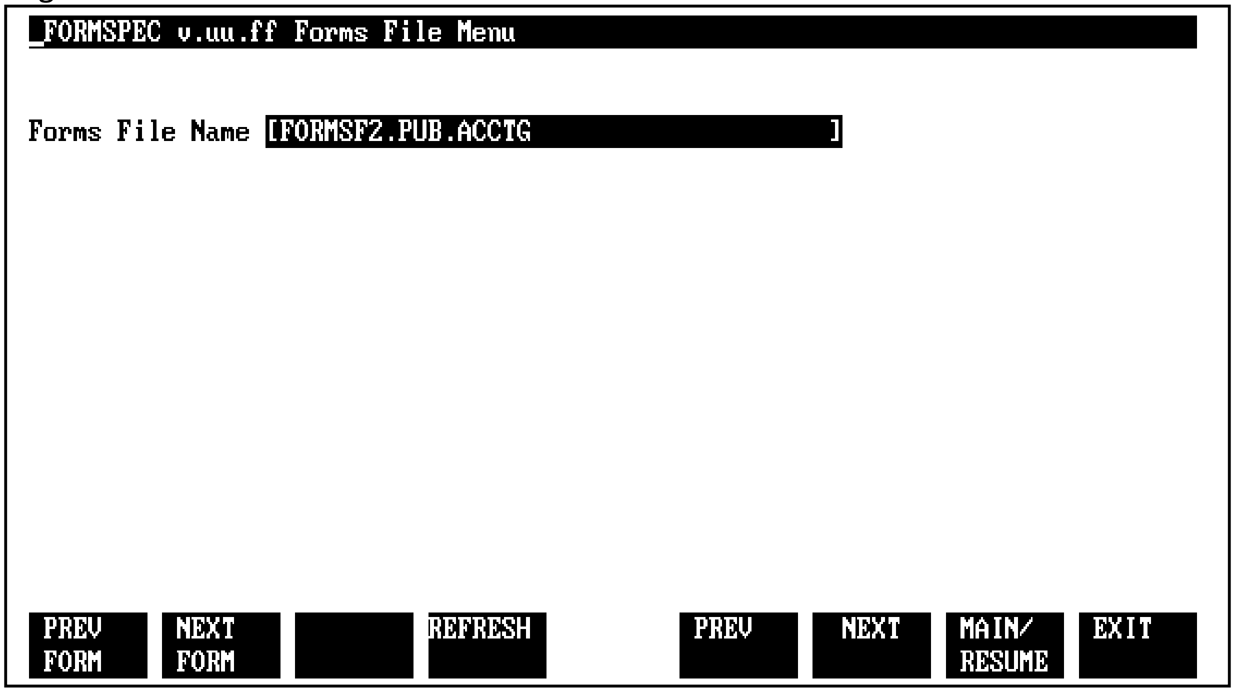
With FORMSPEC, you press **ENTER** to record data; you use function keys to move between FORMSPEC menus and for other tasks, as described below. You can use your terminal editing keys to correct data typed into fields on the FORMSPEC menus; refer to your terminal manual for more information.

Most of the capabilities of interactive FORMSPEC are also available with batch mode FORMSPEC. (See "Using FORMSPEC in Batch Mode", Section 7 for more information.)

Specify a Forms File

The first menu displayed whenever you run FORMSPEC is the Forms File Menu, as shown in Figure 3-1. The letters `v.uu.ff` used for Figure 3-1., and for all the examples shown in this manual, will be replaced by the version number of VPLUS used on your system.

Figure 3-1. Forms File Menu



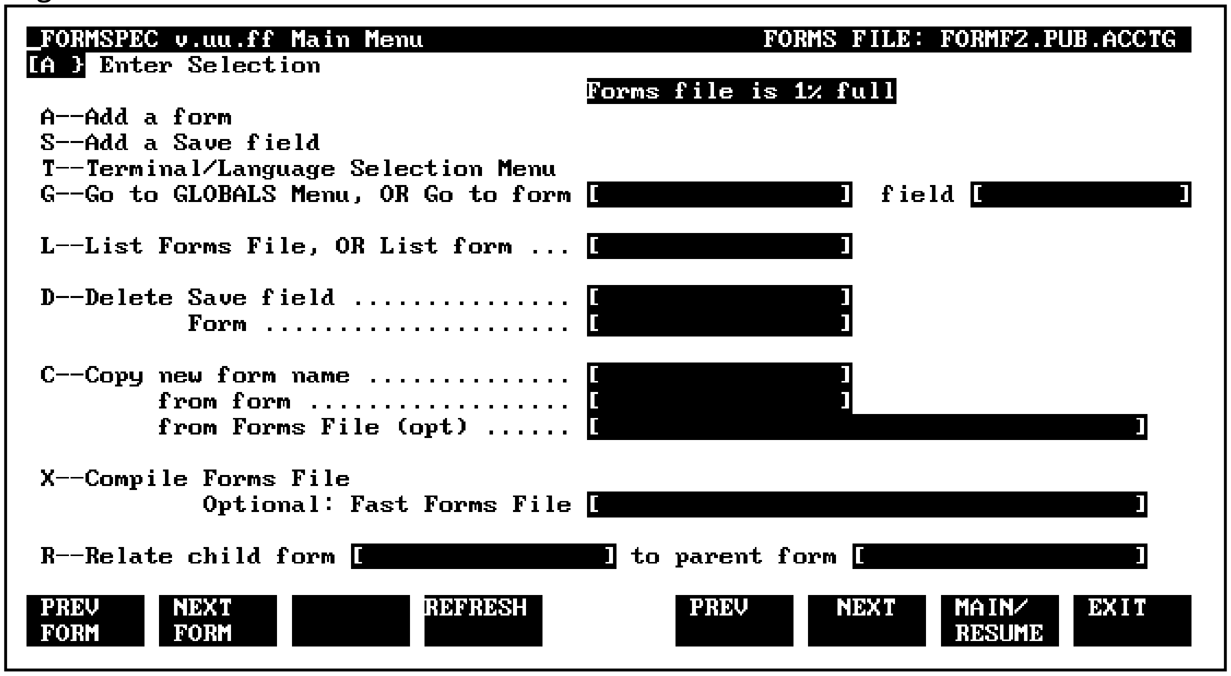
```
_FORMSPEC v.uu.ff Forms File Menu


Forms File Name [FORMSF2.PUB.ACCTG                    ]
```

At the Forms File Menu, you specify the name of the forms file to be created or modified, such as the example FORMSF2.PUB.ACCTG shown in Figure 3-1.; press **ENTER**. If the forms file does not exist, FORMSPEC prompts you to press **ENTER** again to confirm that you want to create a forms file.

Choose an Option

Once you have specified the name of the forms file and pressed **ENTER**, the next menu is the Main Menu, as shown in Figure 3-2. Notice that the top line, called the **window line**, contains useful information, which includes the version number (v.uu.ff), the title of the menu (Main Menu), and the name of the forms file (FORMSF2.PUB.ACCTG).

Figure 3-2. Main Menu



```
 FORMSPEC v.uu.ff Main Menu                         FORMS FILE: FORMF2.PUB.ACCTG
[A ] Enter Selection
                                       Forms file is 1% full
 A--Add a form
 S--Add a Save field
 T--Terminal/Language Selection Menu
 G--Go to GLOBALS Menu, OR Go to form [                 ]  field [              ]

 L--List Forms File, OR List form ... [                 ]

 D--Delete Save field ............... [                 ]
           Form ..................... [                 ]

 C--Copy new form name .............. [                 ]
         from form .................. [                 ]
         from Forms File (opt) ...... [                                      ]

 X--Compile Forms File
            Optional: Fast Forms File [                                      ]

 R--Relate child form [                 ] to parent form [                   ]

 PREV     NEXT              REFRESH           PREV      NEXT      MAIN/     EXIT
 FORM     FORM                                                    RESUME
```

From the Main Menu, you have access to the menus of FORMSPEC with all of their forms design features. You simply type the option desired in the selection box and press **ENTER**. In Figure 3-2. option **A** is specified, which selects "Add a form" and leads to a sequence of menus that allow you to design a form simply and easily. This process is described later under "Ease of Forms Design".

Many of the options, including **L**, **D**, **C**, and **X**, only require information in the fields on the Main Menu in order to be executed immediately. Option **B** lets you create and modify and application-ready buffer for a particular form.

FORMSPEC Function Keys

In addition to the control provided by the Main Menu, the defined function keys, shown in the following table, allow you to select menus either for initial definition or for modification of an existing forms file. The combination of the Main Menu and function keys gives you the ability to change any form, field, or global specification as you define the forms file.

Table 3-1. FORMSPEC Key Labels

Key	Key Action
PREV FORM f1	Display the previous Form Menu. If no forms are defined, a Form Menu with no values is displayed.
NEXT FORM f2	Display the next Form Menu. If the next form is not defined, a Form Menu with no values is displayed.
FIELD TOGGLE f3	Only used in Field Menus to switch between field attributes and the optional processing specifications.
REFRESH f4	Redisplay current menu in its initial state before any specifications were entered or existing specifications modified. Also used to recover in case of unexpected program interruption (refer to Appendix G).
PREV f5	Displays previous menu in sequence of menus (refer to "Menu Sequence").
MAIN/ RESUME f7	Request Main Menu or, if Main Menu displayed, return to menu displayed when MAIN/RESUME was requested.
EXIT f8	Terminate FORMSPEC and return to MPE control.

The function keys used by FORMSPEC should not be confused with the function keys defined by an application or by ENTRY for use while the application is executing, as discussed under "Advanced Forms Design" later in this section. Although the two groups of keys are physically the same programmable keys, their functions differ in most cases.

Menu Sequence

FORMSPEC displays its menus in a predetermined sequence. As each menu is displayed, type in the specifications you want and then press **ENTER**. If you do not want to enter a specification on a particular menu, or you want to skip over one or more menus, you can use the menu sequence control function keys (**NEXT FORM**, **PREV FORM, NEXT**, and **PREV**). These keys allow you to select menus relative to the current menu in order to make changes or additions. Figure 3-3. illustrates the relation between the menus provided by FORMSPEC and the function keys that control menu sequence.

You can move forwards or backwards through the sequence of menus to locate a particular menu by pressing **NEXT** or **PREV**, respectively. If you want to skip the field menus associated with each form, you can use the **PREV FORM** or **NEXT FORM**. An alternate method is to request the Main Menu with **MAIN/RESUME**, and then select an existing menu.

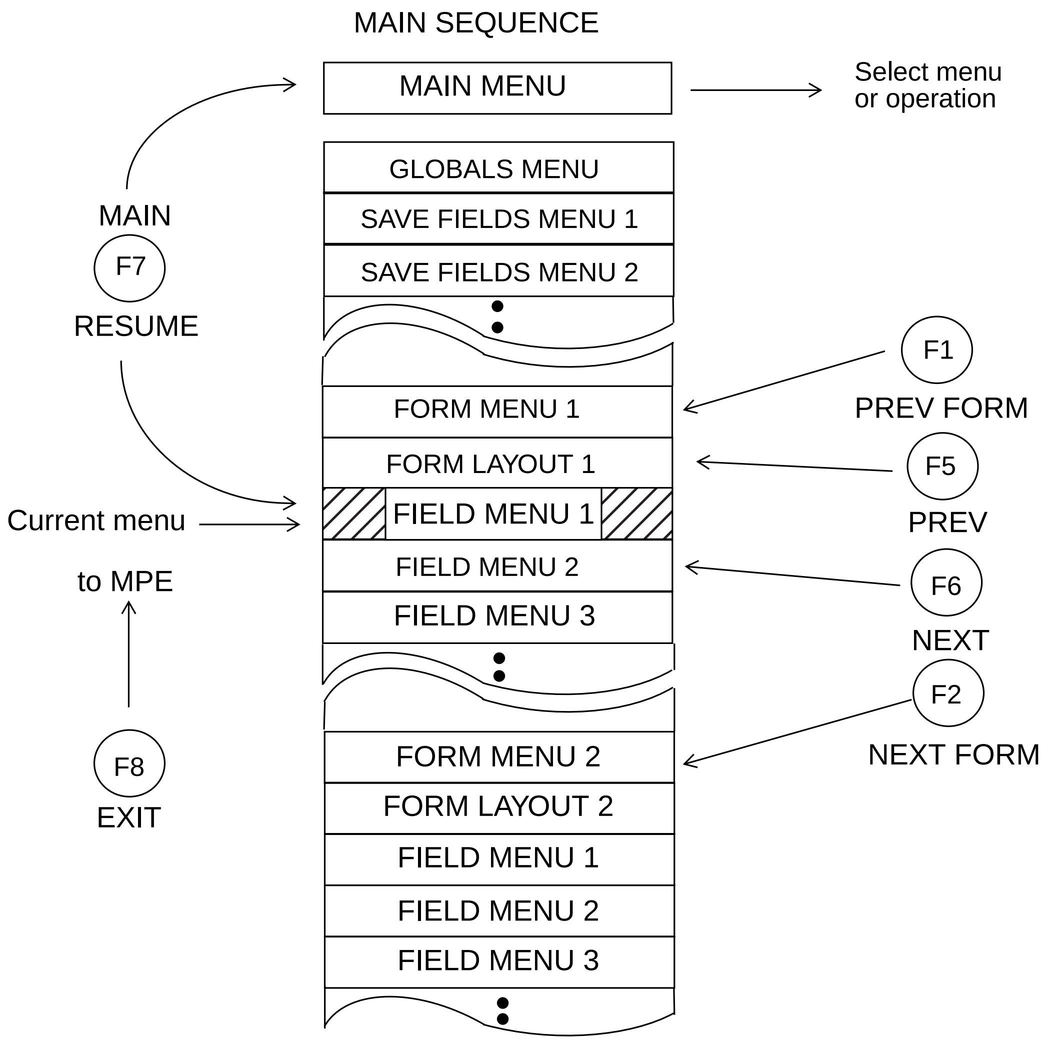
Although it is not technically a menu, the form layout associated with each form is treated as the menu following the Forms Menu when **NEXT** is pressed, and as the menu preceding the first Field Menu when **PREV** is pressed.

The sequence control keys can also be used from the Main Menu. You can go to the Main

Menu at any time from any other menu by pressing **MAIN/RESUME**. You can return to the menu from which you requested the Main Menu by pressing **MAIN/RESUME**. In this case, the menu on the screen when **MAIN/RESUME** was requested is treated as the current menu. If you just entered FORMSPEC and the Main Menu is displayed, pressing **MAIN/RESUME** results in a display of the Globals Menu. From the Globals Menu, **NEXT FORM** causes the first Form Menu to be displayed; **NEXT** causes a Save Field Menu to be displayed.

NOTE The number of Field Menus displayed depends on the number of fields you defined on the screen for the form. The system automatically displays a Field Menu for each field named on the screen and displays the default values for the field. When menus for all fields on a form have been displayed, the Form Menu is redisplayed. If you have no more forms to design, press **MAIN/RESUME** for the Main Menu through which you compile the forms file. Otherwise, continue defining forms by specifying the next form on the Form Menu.

Figure 3-3. Relation between Menus and Function Keys



Creating an Application-Ready Buffer (ARB)

The purpose of the ARB is to allow you to transform screen data into application data and back again by converting data types between screen and application, and by reordering the fields.

Once you have created a form and saved it in the forms file, you can create an application-ready buffer (ARB) for it if you choose. Not every form needs to have an ARB, you can create ARB's for selected forms only. You can also create an ARB for a form at any time after you have created the form itself, and you can delete an existing ARB and generate a new one if you have made extensive changes to the associated form. Fields on the form are mapped one-to-one to the ARB when you select GEN from the ARB Menu. You can rearrange fields on the form using the Screen Design Menu without affecting the ARB, and you can add and delete ARB fields without affecting the form by selecting RES (Restructure) from the ARB Menu.

Before you can generate and ARB, you must define the Data Type Conversions you want to take place between the form on the screen and the ARB. From the Globals Menu, you access the Data Type Conversions menu and define the data conversion defaults. Screen and application data types are listed under "Data Type" further on in this section. Creation of an ARB is described in detail under "Using FORMSPEC to Create an ARB" in this section.

At runtime, the ARB transforms the data as it appears on the screen into a format that the application can use without further manipulation, and converts application data back into a format suitable for display on the screen.

Year 2000 Functionality

VPLUS contains new functionality to address Year 2000 issues. The following sections describe features related to Year 2000 functions.

FORMSPEC Enhancements

FORMSPEC Application Ready Buffer (ARB) processing has been enhanced with the addition of a new ARB data type. This data type can be specified on the DTC Menu for Data Type Conversions from Screen Type to Application Type. The new ARB Type is YYYYMD and is defined as a 8-byte ASCII field containing a date value in YMD order with no separators. Also the year component in the date value has 4 digits. Note that FORMSPEC will ensure a length of 8 for the data type YYYYMD in the ARB. In essence, the value is of the form YYYYMMDD (8 bytes) even though the Type designator on the DTC Menu has been abbreviated to YYYYMD.

Processing Enhancements

Date processing in VPLUS has been enhanced to handle date values with 4-digit year components. Specifically, the following additions have been made.

Date output will be formatted with 4-digit years if the target field is wide enough to receive the date, and the user requests this action. Examples of such output include the SET statement in FORMSPEC. The target field should be at least 10 characters in length to have a 4-digit year output.

VPLUS, by default, will format date output with 2-digit years. To change the behavior as described above, define a JCW called VSETNEXTCENTURY and set bit 15 to 1 (bits are numbered from 0 through 15). It should be noted that this option will be active for fields wide enough to hold the 4-digit year date. For other fields, 2-digit years will be used.

Existing 2-digit years can be interpreted differently to handle dates in the next century. Currently VPLUS handles year components 00-99 as 1900-1999. This scheme of interpretation can be changed. In the new scheme, 00-49 is interpreted as 2000-2049 and 50-99 is interpreted as 1950-1999. This interpretation will be valid through all processing done by VPLUS. For example, this scheme will allow the user to enter 02/29/00 (MDY order) in an existing application and have it processed correctly.

To invoke the above method of date interpretation, define a JCW called VSETNEXTCENTURY and set bit 14 to 1. Note that this method is used for processing only 2-digit year dates and not 4-digit year dates. Note also that the display of 2-digit year dates is not affected by this enhancement. This means that 12/11/10 (MDY order) can refer to 1910 in one application and 2010 in another application.

NOTE When defining the JCW VSETNEXTCENTURY, set all unused bits to 0.

Programmable Defaults for Field and Data Types in FORMSPEC

FORMSPEC version B.06.07 has been enhanced to provide the capability to set the default Field Type and Data Type of a form. This capability can be used to reduce the time taken to complete the design of a new form. Two new fields for user input have been added to the Form Menu screen of FORMSPEC. One is for the default Field Type, and the other is for the default Data Type.

On the Form Menu, the user typically specifies the name of the form and other characteristics of the form. In addition now, the user can specify the default Field Type and Data Type for the form. For a list of the allowable values for the Field Type and Data Type, refer to chapter 2 of this manual. All fields created newly for the form will have the Field Type and Data Type set to the default values supplied on the Form Menu.

Notes

- The Field Type and Data Type values supplied on the Form Menu are applicable only to fields created for the form thereafter. This means that if these default values are changed for an existing form, the types of the existing fields are not changed. For a new form, the default values apply to all the fields created for the form.
- Cloned forms have their screen already designed; hence, the values on the Form Menu do not apply to such forms.
- The Field Type and Data Type can be specified independently. This means that either, or both, fields on the Form Menu can be blank.
  - — If the Field Type is blank, all newly created fields for the form will have a Field Type of “O.”
  - — If the Data Type is blank, all newly created fields for the form will have a Data Type of “CHAR.”

Ability to Define More than 52 Single Character Fields per Form

This enhancement allows more than 52 single character fields to be defined for a Form. Currently the field tag which is defined for a field at screen design (creation) time should begin with an alphabetic character. This enhancement will allow several other characters to be used for the field tag, thus increasing the number of single character fields that can be defined for a Form.

The following are the other characters which can be used for the field tag (defined at screen creation time):

Digits	`0,1,2,3,4,5,6,7,8,9`
Specials	`@,#,$,%,&,*,-,+,<,>,/,\,!,|,=,?,;,_`

Field Menu Initialization

In the case where one of the characters listed above (a Digit or a Special) is used in a single character field, the Field Name will be blanked out when the Field Menu is invoked for the field the first time. The user should input a new name for the field in this case. With this enhancement, a maximum of 80 single character fields can be defined for a Form.

Compatibility Issues

Forms created with previous versions of FORMSPEC can be used with this version without any modifications.

Forms created with this version of FORMSPEC can be used with existing applications without any modifications.

Forms created with this version of FORMSPEC can be used with previous versions of FORMSPEC. However, some extra characters may appear near the bottom of the Form Menu screen which should be ignored. These extraneous characters appear only once, the first time the Form Menu is invoked for the form, and they do not affect the functionality in any way.

Terminating FORMSPEC

You can terminate operation of FORMSPEC at any time by pressing EXIT. This returns you to MPE control which then issues the MPE colon prompt (:).

However, if you have made any modifications and have not compiled the forms file, FORMSPEC issues a warning. By pressing EXIT again you terminate operation of FORMSPEC without compiling the file. Refer to "Forms File" for more information on compiling the forms file.

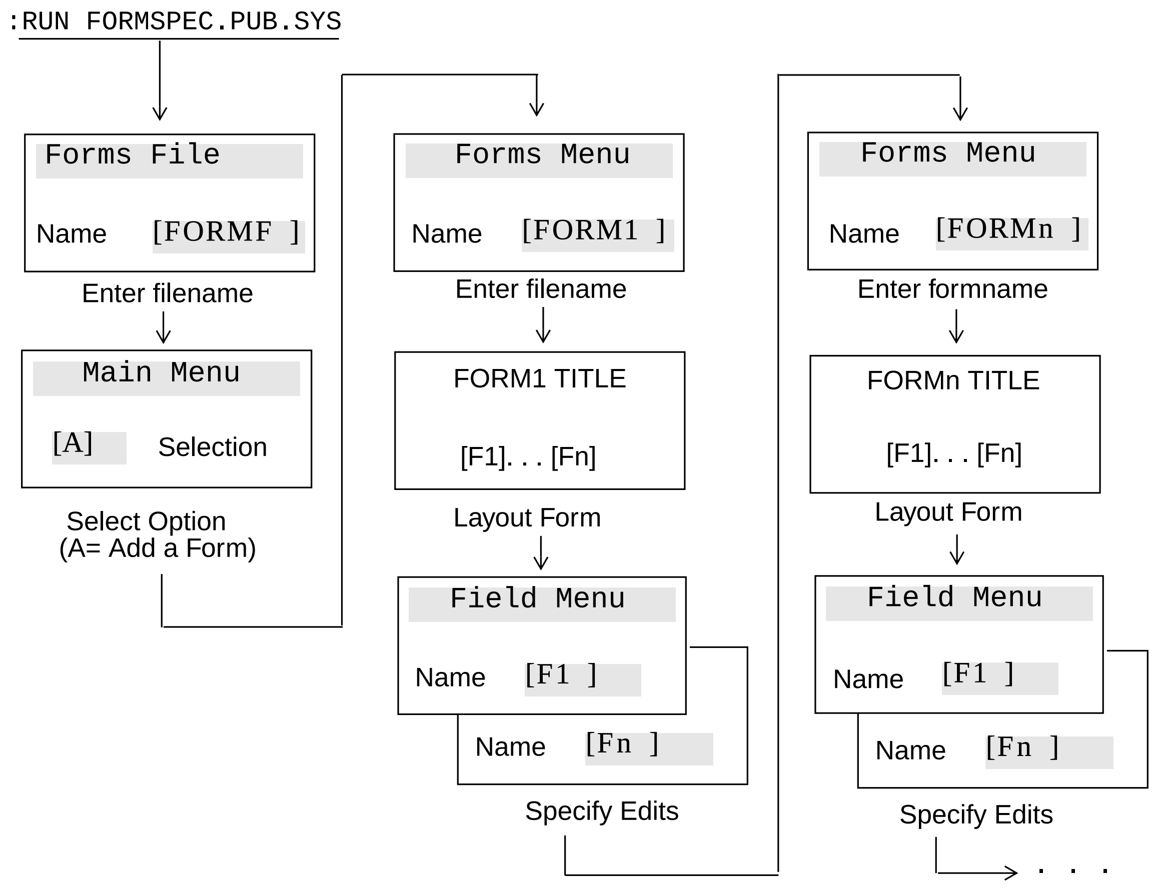
Unexpected Program Interruption

In case of unexpected interruption due to hitting **BREAK** or a terminal power failure, control returns to MPE. Refer to Appendix G for the steps to recovery from such a situation. Once you have recovered, the menu will be cleared to initial or previously entered values. To ensure against damage to the file, reenter the information on all menus pertaining to the form you were creating or modifying at the time of the program interruption.

EASE OF FORMS DESIGN

Once you have specified a forms file and the Main Menu is displayed, you are ready to use FORMSPEC to create forms. Whenever you create a new forms file, your first task is always to add a form; if you are modifying an existing file, you may want to add a form. In either case, you follow the sequence of menus shown in Figure 3-4. Refer to the menu descriptions later in this section for more information on all the FORMSPEC menus.

Figure 3-4. Sequence of Menus for Form Design

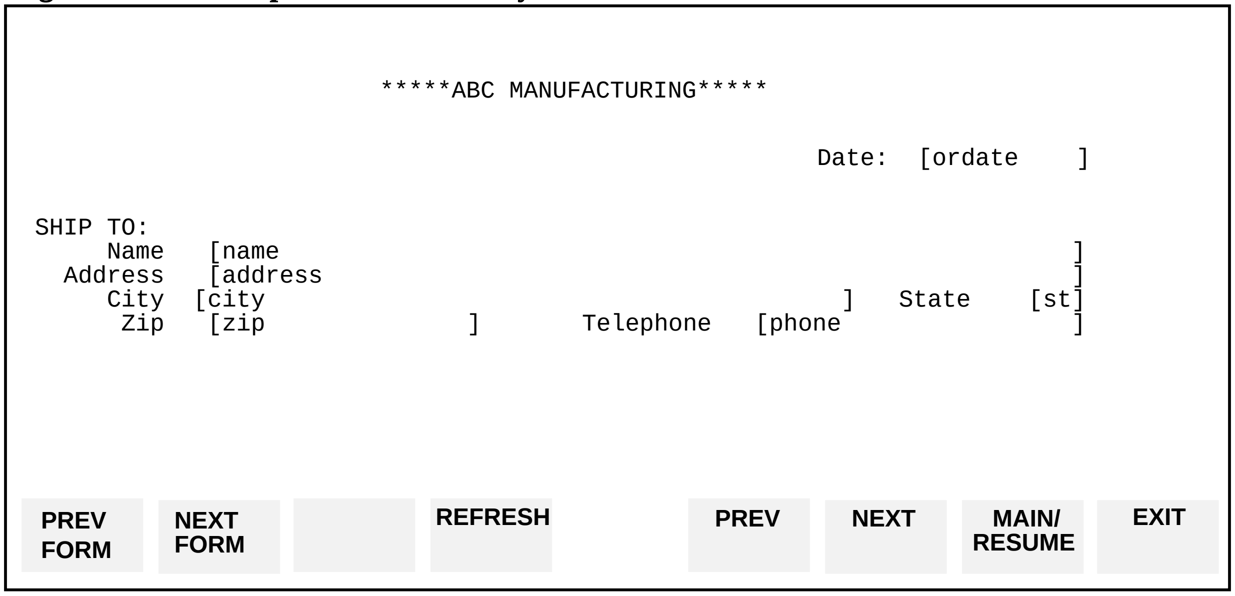


A Form Menu, shown later in Figure 3-10., allows you to define the characteristics of each form, such as the form name and what form sequencing options to use. Refer to "Understanding Form Sequencing" below.

Form Layout

The Form Menu is followed by a blank screen on which you design the layout of the form as it will appear on the screen. An example of a form layout is shown in Figure 3-5. This layout is easy to draw on the screen — and easy to change at this stage or later. You use your terminal editing keys to layout the fields and text of your form on the screen; refer to your terminal manual for more information.

Figure 3-5. Example of a Form Layout


```
                           *****ABC MANUFACTURING*****

                                                              Date:  [ordate    ]


 SHIP TO:
      Name   [name                                                         ]
   Address   [address                                                      ]
      City  [city                                       ]   State    [st]
       Zip   [zip              ]       Telephone   [phone                  ]
```

Each layout consists of two kinds of information:

Fields	Areas, delimited on the form, into which data will be entered by or displayed to the user. The maximum number of fields allowed on anyone form is 128.
Text	The headings and other displayed information that appears on the form but is never altered during execution.

You must distinguish between these two kinds of information within the form layout using field delimiters and field tags to indicate the fields.

Field Delimiters

The data fields are delimited by brackets (**[ ]**), by **ESCAPE** followed by brackets, or by a combination of these. Pressing **ESCAPE**, then bracket, prevents the brackets from being displayed on the screen.

Printed Delimiters. If you delimit the data fields with brackets, they are not included in the length of the field, but they do take on the display enhancements assigned to the field. Brackets make it easy to see where a field begins and ends during definition, but they take up space on the form. If you must concatenate two data fields, you should use nonprinting delimiters to delimit the fields.

Nonprinting Delimiters. You can also delimit fields by pressing **ESCAPE** followed by:

the left bracket (**[**) OR the right bracket (**]**).

The advantage of these delimiters is that they take up no space on the form and thus can be used to delimit contiguous fields. The disadvantage of using these keys is that they are not displayed and therefore do not show up during form design.

Mixing Printing and Nonprinting Delimiters. To use one printing and one nonprinting delimiter to fix the boundaries of a field, use the delimiters as follows:

If the printing delimiter is to come first, use

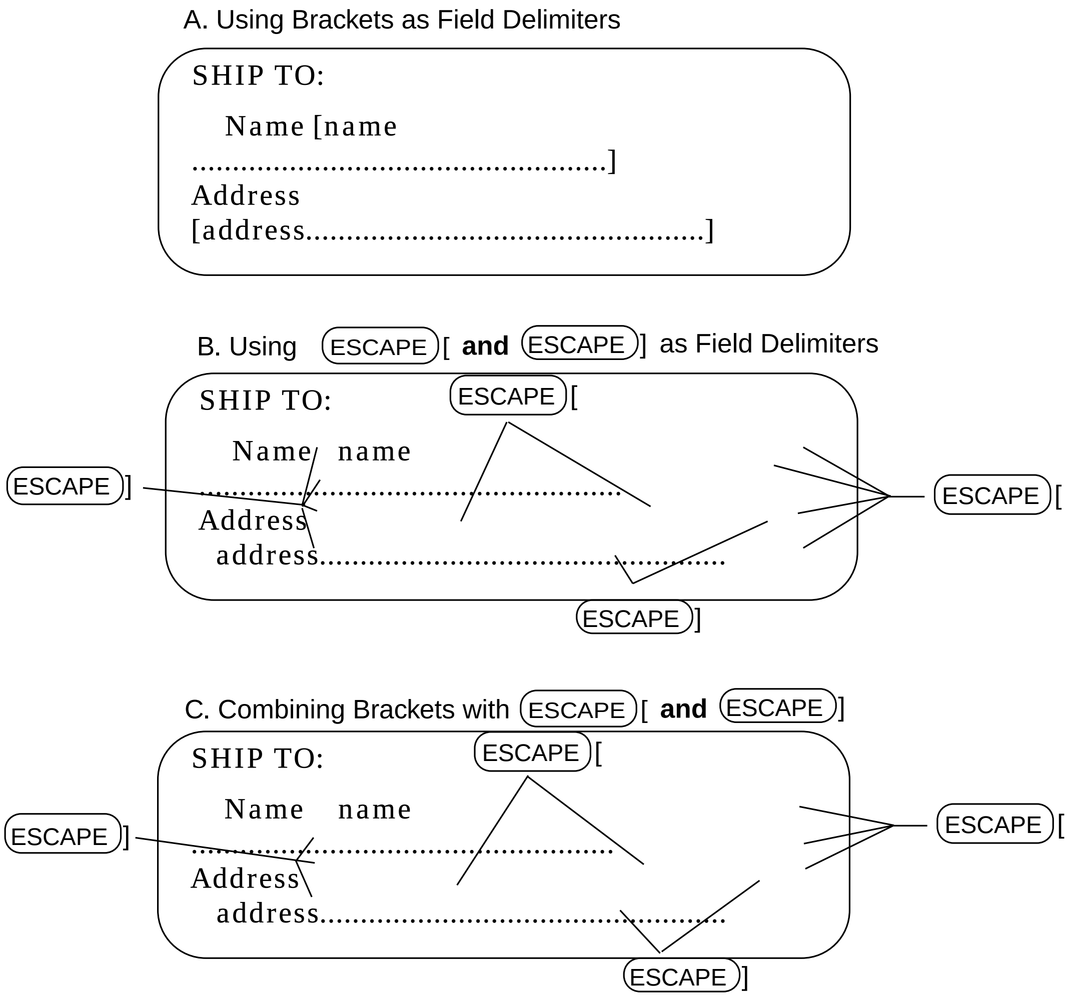
[**ESCAPE** [*fieldtag*. . . **ESCAPE**]

If the printing delimiter is to come last, use

ESCAPE [*fieldtag*. . .**ESCAPE**]

In the case where a field begins with **ESCAPE** [and ends with], the terminal will insert **ESCAPE**] if a following field also starts with **ESCAPE** [. As a result VPLUS will reject the form layout. This terminal feature can be avoided by properly using the above rules for mixing delimiters. Refer to Figure 3-6. for examples of different ways to layout a form.

Figure 3-6. Examples of Form Layouts


A. Using Brackets as Field Delimiters
SHIP TO:
Name [name
....]
Address
[address..]
B. Using ESCAPE [and ESCAPE] as Field Delimiters
SHIP TO:
ESCAPE [
Name name
ESCAPE]
....
Address
address...
ESCAPE [
ESCAPE]
C. Combining Brackets with ESCAPE [and ESCAPE]
SHIP TO:
ESCAPE [
Name name
ESCAPE]
....
Address
address...
ESCAPE [
ESCAPE]

Field Tag

Regardless of the technique used to delimit the fields, each field must be identified by a "field tag". This tag consists of (USASCII only) letters of the alphabet (uppercase or lowercase), digits, or the underline (). The first character must be alphabetic. Since the field tag must be specified within the field delimiters, it is limited to a length less than or equal to the number of characters in the field. Thus, if you have a one-character field, its tag may not have more than one character. However, the field tag is also used for the field name. You may change any field name on the Field Menu. Thus, you can give a one-character tag a longer field name when the Field Menu for the field is displayed.

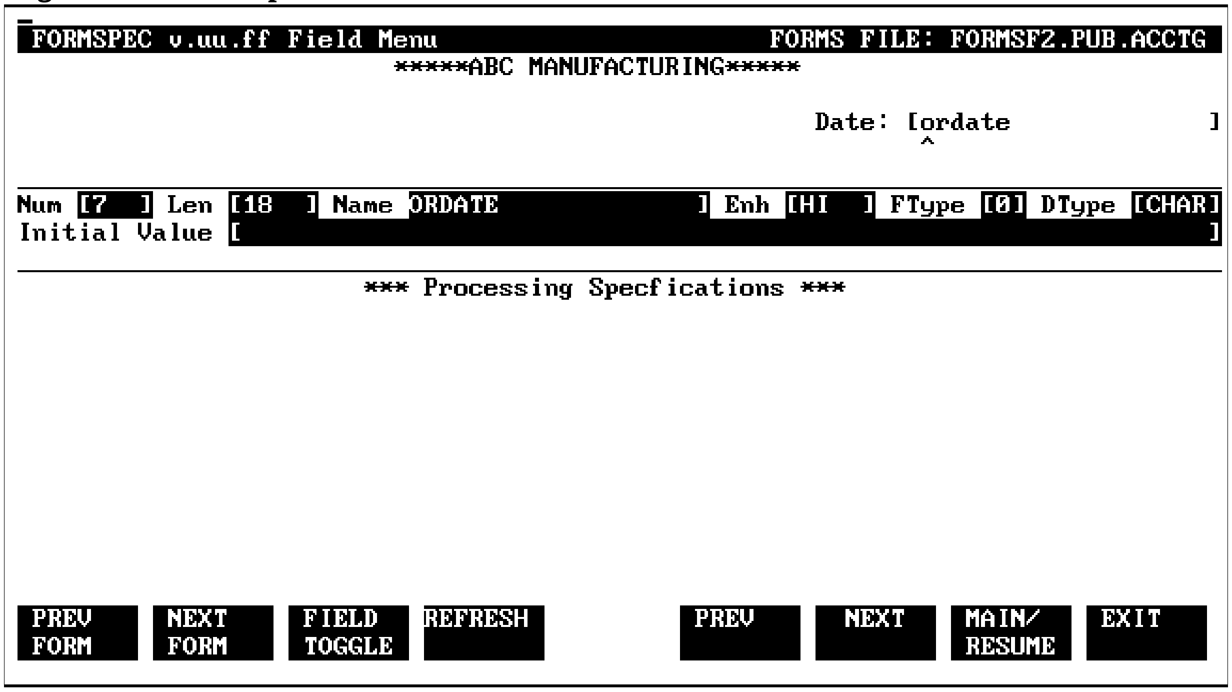
Note that for the field tag, uppercase letters differ from lowercase letters. Thus, `f1` and `F1` are two different tags. All other names used by FORMSPEC make no distinction between uppercase and lowercase letters, but shift all letters to uppercase. Since each tag is upshifted when used as the default field name, tags that differ on the form may result in identical field names. When this occurs, you must rename one of the identical field names on a Field Menu. Each field tag must be unique before it is upshifted, thus, a form may have up to 52 one-character fields.

You can fill up the field with dots (periods). This gives you a visual representation of field size while you are designing the field. Using dots in the field is particularly useful when the field is delimited by **ESCAPE** [and **ESCAPE**] since these delimiters are not displayed. The dots do not show up when the form is displayed by the application.

Defining the Fields

After defining the form layout, FORMSPEC displays a Field Menu for each field, such as the example shown in Figure 3-7.

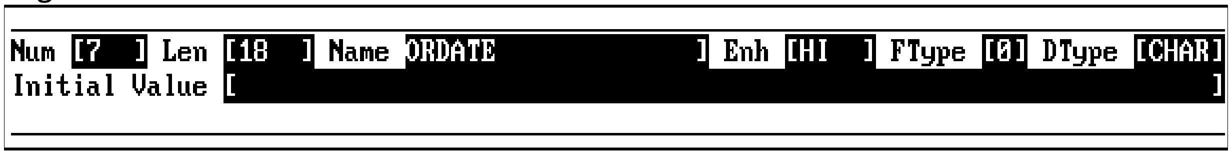
Figure 3-7. Example of a Field Menu


```
 FORMSPEC v.uu.ff Field Menu                     FORMS FILE: FORMSF2.PUB.ACCTG
                       *****ABC MANUFACTURING*****

                                                          Date: [ordate              ]
                                                                  ^

Num [7  ] Len [18  ] Name ORDATE                ] Enh [HI  ] FType [O] DType [CHAR]
 Initial Value [                                                                 ]

                       *** Processing Specfications ***
```

Each Field Menu displays the portion of the form containing the current field. Compare Figure 3-5. and Figure 3-7.; notice the portion of the form layout of Figure 3-7. that is displayed in the Field Menu in Figure 3-5. The current field is indicated by a caret (^) under the field tag. In the example in Figure 3-7., the current field is labeled `Date:` with the field tag of `ordate.`

The information you enter on the Field Menu is divided into two categories: field attributes and processing specifications. The field attributes are the two lines of fields below the portion containing the current field. For example:

Figure 3-8. The Field Attributes



```
Num [7  ] Len [18  ] Name ORDATE              ] Enh [HI  ] FType [O] DType [CHAR]
Initial Value [                                                               ]
```

Only the field attributes are discussed here. The processing specifications are described in Section 4. If you can use the field attributes without change, the entire form design is complete at this point. If not, you can specify any simple edits, as described below, or, if you want to use the FORMSPEC processing specifications, described in Section 4, you can enter appropriate processing specifications on the Field Menu. As shown in Figure 3-4., the form design sequence can be repeated until all forms in the file are defined.

Field Number

Although your form may have a maximum of 128 fields, a number between one and 256 is assigned by FORMSPEC to each new field. This number is assigned to the field and is not changed even if other field characteristics are changed. If, however, the field tag is changed, this effectively deletes the field associated with the old tag. The field number is deleted along with the field, and a new number is assigned to the field associated with the new tag. The field numbers of a form can be renumbered in screen order with the FORMSPEC batch mode command RENUMBER, as described in Section 7.

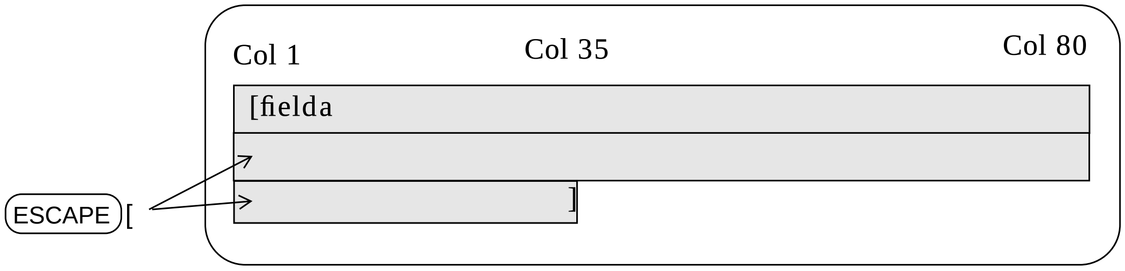
Field Length

This is the length of the field as determined by the number of characters entered between the field delimiters during form layout. The length of a field cannot be changed on the Field Menu. If you want to change field length, you must display the form layout and actually change the field on the form. If you change the field length on the form, the new length is automatically reflected in the Field Menus.

If you want to design a field that is longer than one line, you start the field with a bracket (or **ESCAPE**[) and terminate it with a closing bracket (or **ESCAPE**]). At the beginning of each intermediate line of the field, you enter an **ESCAPE** [.

For example, as shown in Figure 3-9., consider a field that extends across three lines.

Figure 3-9. Example of a Field Extending over Several Lines



Assuming the first line of the field contains 79 characters, the second line contains 80 characters, and the third line contains 34 characters, the entire field is 193 characters long. This count excludes the printing brackets which delimit the beginning and the end of the field.

Field Name

The field tag assigned during form design is shifted up to all uppercase letters and becomes the field name. You can enter another name in this field. For example, you may want a longer name than would fit in the field, or if two tags are no longer unique when shifted to all capitals, you can rename one of them here. In any case, the name in this field (an upshifted tag or a new name) identifies the field in subsequent references. It must not be one of the reserved words listed in Table 3-2.

Table 3-2. FORMSPEC Reserved Word List

ALL	FILL	LOCALEDITS	SET
APPEND	FINISH	LT	STDCHAR
BARCODE	FREEZE	MAGSTRIPE	STOP
CAD	GE	MARKS	STRIP
CDIGIT	GT	MAT	THEN
CENTER	HOLES	MATCH	TO
CFORM	IF	MFR	TRAILING
CHANGE	ILV	MINLEN	TYPEV
CLEAR	IN	NE	UPC
COD	INIT	NFORM	UPSHIFT
CONFIG	125	NIN	$EMPTY
CUT	139	NOCUT	$END
DEVICE	JUSTIFY	NONE	$HEAD
DISPLAY	KEYBOARD	NOREPEAT	$LENGTH
EAN	LARGECHAR	OF	$REFRESH
ELSE	LE	PRINTER	$RETURN
EQ	LEADING	RELAY	$STATE
FAIL	LEFT	REPEAT	$TODAY
FIELD	LIGHT	RIGHT	

You can enter any name up to 15 characters long. Like other FORMSPEC names, it must be USASCII and start with an alphabetic character. It may be followed by uppercase or lowercase letters (A-Z), numbers (0-9), or an underline ().

Default The uppercase field tag.

Display Enhancement

You can change the enhancement for the particular field with any combination of the codes shown in Figure 3-3. Up to four of the available enhancements may be used at one time. The data capture devices do not support display enhancements.

Table 3-3. Display Enhancement

Enhancement	Code
Half Bright	H
Inverse video	I
Underline	U
Blinking	B
Security	S *
Color	1-8 *
None	NONE

Security and color enhancements are only available on terminals with the security or color feature, as listed in Appendix G. Refer to "Using Terminal Features" for more information on security and color.

Field Type

The field type is specified as D, R, O, or P to indicate one of the following options:

Display(D)	Field cannot be modified by the user. Intended for display only, the field can receive data as specified by processing specifications (see Section 4). It is a protected field on the form, but data is written to display only fields exactly as if it had been entered by the user.
Required(R)	Field cannot be left blank. User must enter nonblank values in field.
Optional(O)	Field may be left blank by user. Edit checks and field phase processing specifications for the field are skipped if the field is left blank.
Process(P)	Exactly like an optional field, except that edit checks are performed and field phase processing specifications are executed even if the field is blank.
	Default O (optional)

Required, optional, and process fields are treated as "input" fields. An input field is one in which the user can enter or change data, as opposed to a display only field which is protected from user input. Note that initial values may be displayed in any field, but such values can be, and usually will be, changed by the user.

Data Type

There are two sets of data types, the screen data types and the ARB data types. If you are creating an ARB for a form, the application needs to know the ARB data types. Each field on the form and the ARB must be specified as one of three data types: character, numeric, or date. Based on the data type, FORMSPEC can determine the basic validity of the data entered, how it is formatted for data movement, and the type of operations that can be performed on the field.

Screen data types impose format and edit rules for data entered on the screen. ARB data types determine how that data will be interpreted by the application. Conversions from screen data type to ARB data type and vice versa occur automatically at runtime (see VGET/PUTBUFFER in section 7).

Screen Data Type. Valid screen data types are: CHAR, NUM*(n)*, DIG, IMP*n*, MDY, DMY, and YMD. Figure 3-4. illustrates how each numeric data type interprets an entered value on the screen. The data types are described in the following paragraphs.

Default CHAR

Character Type. CHAR

Data entered in the field is assumed to be a string of any characters. No validity checking is performed on data at time of entry. No arithmetic operations can be performed on data of this type. The user can enter any characters in a CHAR type field. (As noted above, CHAR is the default data type.)

For example, `$12.59, A-15-75,` and `**123**` are all legitimate entries.

Numeric Types. NUM[*n*]

Data entered in a field of this type must be numeric. The maximum number of decimal places can be indicated by the optional digit, *n*, where *n* is a value between 0 and 9. If you omit the value, *n*, the data item is assumed to have a floating decimal point.

Validity checking is performed on this data type. An optional leading sign (+ or -) is allowed. Commas are allowed, but if included they must be correctly placed. An optional decimal point may be included in the data. When Native Language Support is used, the symbols used to indicate thousands and decimals are language-dependent. For more information on Native Language Support, see Section 8.

The user must enter numeric data in this field. If *n* specifies a maximum number of decimal positions, the user must include a decimal point when the value has decimal places, and may omit the decimal point when it does not.

For example, in a NUM2 field, the user can enter a value with no decimal places (such as `500`) or a value with one decimal place (such as `5.5`) or a value with two decimal places (such as `5.95`). Commas and a sign may also be included. For example, the following are legitimate entries for a NUM2 field: `1,390, -327.00, +100,000.00` and so forth; but the value `5.678` is disallowed as it has too many decimal places.

`DIG`	Data entered in a field of this type must be a positive integer. No sign, no commas, and no decimal point can be included. As with NUM data, validity checks are performed on this data type. For example, `10030, 2` and `307` are all legitimate entries in a DIG field, but `10.5, 100,000` or `-10` are not.
`IMP` *n*	A value entered in an IMP field has an assumed decimal point. The assumed decimal point is *n* places from the right of the value. (Note that the value, *n*, must be specified.) Although permitted, a decimal point should not be entered in this type field, but commas and an optional leading sign (+ or -) are allowed. As with NUM type data, validity checks are performed on data of this type. For example, if the data type is IMP2, and the user enters the value `500`, the value is treated as `5.00`. With Native Language Support, the symbols accepted for thousands and accepted or required for decimal indication are language dependent. For more information on Native Language Support, see Section 8.

Table 3-4. How Each Numeric Data Type Interprets Entered Values

Entered Value	Interpreted Value (Based on data type)				
	NUM2	**IMP2**	**NUM**	**NUM0**	**DIG**
10.95	10.95	10.95	10.95	(error)	(error)
1095	1095.00	10.95	1095	1095	1095
10.9	10.90	10.90	10.9	(error)	(error)
10.956	(error)	(error)	10.956	(error)	(error)
-100	-100.00	-1.00	-100	-100	(error)
1,000	1,000.00	(error)	1,000	1,000	(error)
1,00000	(error)	1,000.00	(error)	(error)	(error)

Date Types.

`MDY`	A date entered in an MDY field must be in the order: month, day, year. The data can be entered in any format, such as `FEB 6, 1986` or `02/06/86` or `FEBRUARY 6 86` and so forth.
`DMY`	A date entered in a DMY field must be in the order: day, month, year. It can be any format, such as `2 MAR 1986` or `02-03-86` or `2/3/86` and so forth.
`YMD`	A date entered in a YMD field must be in the order: year, month, day. It can be in any format such as `1986, APRIL 12` or `86/4/12` or `86-04-1` or `860402`.

The entered data is checked by VPLUS for correct format and that it is a valid date. Arithmetic operations are not allowed on date fields.

Native Language Support does not affect the order of the date field. It does, however, accept the names of months and their abbreviations for each native language at run-time. For more information on Native Language Support, see Section 8.

To illustrate how the three date type specifications interpret entered dates for NATIVE-3000, Figure 3-5. shows legal dates followed by Figure 3-6. with dates that would be diagnosed as illegal.

Table 3-5. Valid Dates

MDY	DMY	YMD
`February 7, 1986`	`7 February 1986`	`1986, February 7`
`FEB 7 1986`	`7 FEB 1986`	`1986 FEB 7`
`02/07/86`	`07/02/86`	`86/02/07`
`2/7/86`	`7/2/86`	`86/2/7`
`02-7-86`	`07-2-86`	`86-2-07`
`2 7 86`	`7286`	`8627`
`020786`	`070286`	`860207`

Table 3-6. Invalid Dates

MDY	DMY	YMD	Reason
`Febrary 7, 1986`	`7 Febrary 1986`	`1986, Febrary 7`	Misspelled
`FEBR 7 1986`	`7 FEBR 1986`	`1986 FEBR 7`	Four letter abbreviation
`2786`	`7286`	`8627`	Must be a two-digit month, day when no separator is included.

ARB Data Types The valid choices for ARB data types are: `CHAR, INT, DINT, REAL, LONG, SPACK`*n*`, PACK`*n*`, SZONE`*n*`, ZONE`*n*, and `YYMMDD`. Data type and length may differ from screen to ARB. The ARB data types include some that are language-specific. For example, `SPACK`*n* and `PACK`*n* correspond to `COBOL COMP-3; SZONE`*n* and `ZONE`*n* correspond to `COBOL` signed display numeric and unsigned display numeric respectively; and `INT` and `DINT` correspond to `COBOL COMP`. For more information on COBOL data types, see the *COBOLII/3000 Reference Manual*. The default data type and length for an ARB field are derived from the data type and length of the corresponding field on the associated form and the Data Type Conversion record.

Figure 3-7. sets out recommended guidelines for data type conversions from screen to ARB and back.

Table 3-7. Recommended Data Type Conversions

Screen Data Type —>	**ARB Data Type**
`CHAR`	`CHAR` only
`YMD, DMY, MDY`	`YYMMDD, CHAR`
`DIG`	any except `YYMMDD, CHAR`
`NUM[n], IMPn`	any except `YYMMDD, CHAR`; "n" truncated if `INT, DINT`
ARB DATA Type —>	**Screen Data Type**
`CHAR`	`CHAR` only
`YYMMDD`	`MDY, YMD, DMY, CHAR, DIG`
`INT, DINT`	`DIG, NUM[n], IMPn`; positive only if `DIG`
`REAL, LONG, PACK, ZONE`	`DIG, NUM[n], IMPn`; if `DIG` - "n" truncated, positive only

FORMSPEC will ensure the following length specifications for ARB data types:

```
YYMMDD length = 6
INT    length = 2
DINT   length = 4
REAL   length = 4
LONG   length = 8
```

YYMMDD is defined as a 6-byte ASCII field containing numeric data with no separators, in YMD order; for example, 860419.

The designer is responsible for making legitimate data conversions. The three critical factors are data type, value, and length. The following examples illustrate their importance.

- Data Type: Runtime errors may arise from specifying a CHAR/DATE source conversion to a numeric destination, or a CHAR source to a DATE destination.
- Value: The ARB data type INT may have a screen type of CHAR. This works if the field is for display only, but if it is an entry field, the user could input ABC unless the designer has taken steps to prevent it.
- Length: Data may be truncated if the screen data type DIG is specified for a field of length 10 and the ARB type is INT (length 2 bytes, equal to 1 HP 3000 word). Both COBOL and Pascal can store numbers in the range -32768 to 32767 in one word.

NOTE These are recommendations only, and are not enforced by FORMSPEC edits. The programmer may set up the application code to handle non-standard conversions.

Figure 3-8. shows the valid screen data types and their corresponding values for COBOL, Pascal and FORTRAN.

Table 3-8. Valid Screen Data Types

ARB Data Type	COBOL	Pascal	FORTRAN
CHAR	PIC X (_)	Packed array CHAR [.._]	CHAR
YMD	as above	as above	as above
DMY	as above	as above	as above
DIG	as above	as above	as above
NUM[n]	as above	as above	as above
IMP[n]	as above	as above	as above
Note: [n] represents the number of decimal digits			

Figure 3-9. shows valid application data types and their values in COBOL, Pascal and FORTRAN.

Table 3-9. Valid Application Data Types

FORMSPEC	COBOL	Pascal	FORTRAN
CHAR	X	Packed array CHAR [.._]	CHAR *_
YYMMDD	X(6)	as above	CHAR *6xx
ZONEn	9(_)Vn		
PACKn	9(_)Vn COMP-3		
SPACKn	S9(-)Vn COMP-3		
REAL		REAL	REAL
LONG		LONG	Double Precision
INT	S9(4) COMP	Subrange -32768..32767	Integer **2
DINT	S9(4) COMP	Integer	Integer **4

Initial Value

You may specify an initial value for the field. This value will be displayed in the field when the form is first displayed at the terminal. It is also displayed when REFRESH is pressed during data collection in ENTRY and the form is cleared to its initial values.

The value entered here is treated like user input. That is, it must match the data type of the field and must not be longer than the field length.

(Refer to Section 4 for a discussion of how FORMSPEC uses these data types to perform validity checks on data entered in the fields, and how data is treated during movement from one field to another.)

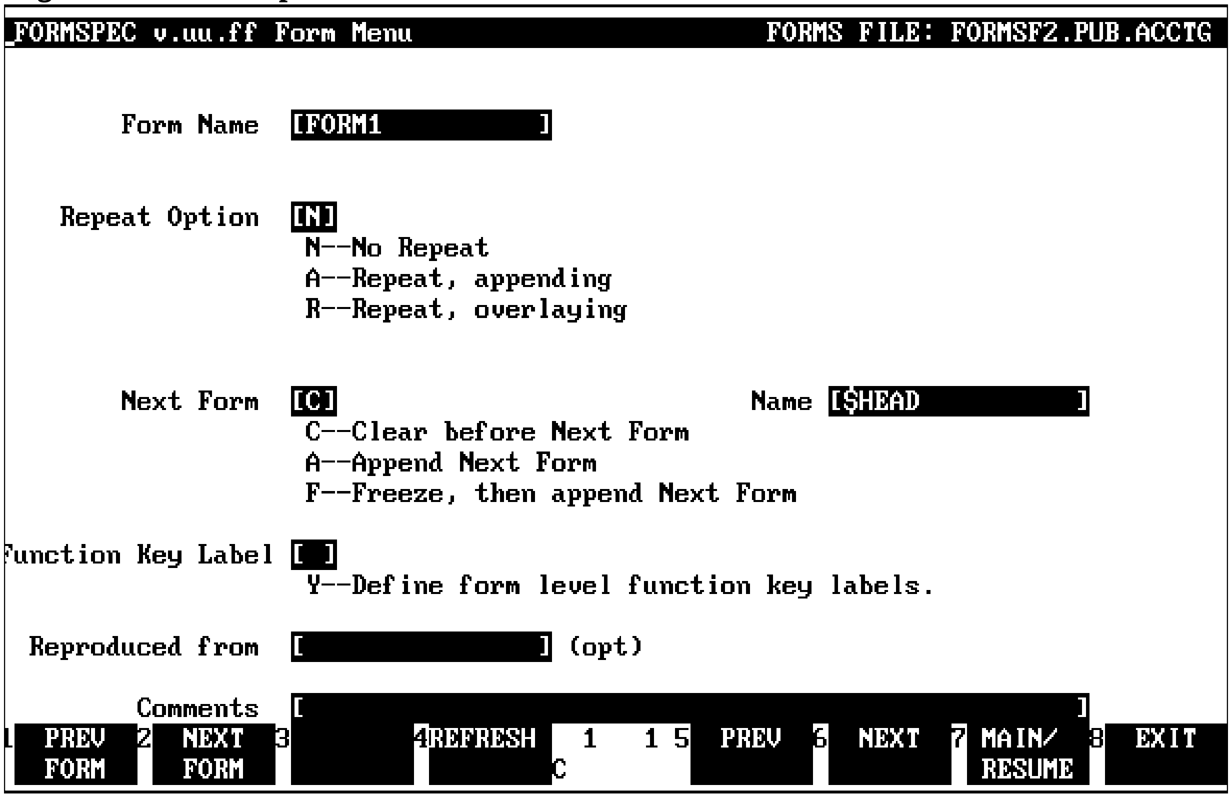
If Native Language Support is used, you must specify the initial value in NATIVE-3000.

The value will appear with the conventions of the native language at run-time. For more information on Native Language Support, see Section 8.

Understanding Form Sequencing

As shown in Figure 3-4., the Form Menu is the first menu in the sequence of menus used to define a form. On the Form Menu, as shown in Figure 3-10., you specify a name for the form and the form sequencing options to use (Repeat Option and Next Form The form sequening options are described below. the other fields pertain to advanced features such as “Form Families” discussed later in this section.

Figure 3-10. Example of a Form Menu

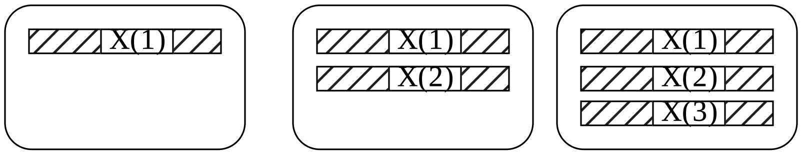


```
 FORMSPEC v.uu.ff Form Menu                    FORMS FILE: FORMSF2.PUB.ACCTG


       Form Name [FORM1           ]


   Repeat Option [N]
                  N--No Repeat
                  A--Repeat, appending
                  R--Repeat, overlaying


       Next Form [C]                          Name [$HEAD           ]
                  C--Clear before Next Form
                  A--Append Next Form
                  F--Freeze, then append Next Form

Function Key Label [ ]
                  Y--Define form level function key labels.

 Reproduced from [                ] (opt)

        Comments [                                                  ]
1 PREV   2 NEXT   3          4REFRESH  1    1 5  PREV  6  NEXT  7 MAIN/   8  EXIT
  FORM     FORM                       C                          RESUME
```

The Repeat Option

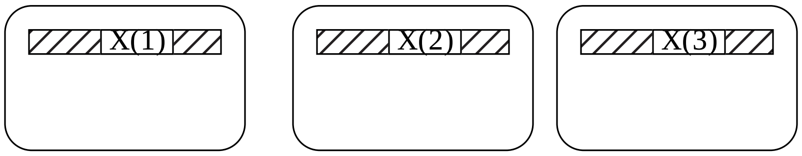
When the forms in a forms file are displayed, a form may be repeated and appended to itself (A); it may be repeated overlaying the previous display of itself (R); or it can be a non-repeating form that is displayed once (N).

To illustrate how the three choices of the repeat option work, assume that form X is a repeat/append form (A). This form is displayed, the user types in data, and presses **ENTER**. The form with the data remains on the screen, and the same form with no data (except initial values) is displayed Immediately below the form with data. The next time **ENTER** is pressed, the form is displayed a third time immediately below the second form. This continues until the repeat option is changed.



Appended forms are particularly useful when the form is a single line that has an indeterminate number of iterations. An order entry blank, for instance, could be designed as a repeat/append form.

A form that repeats without the append option (R) is cleared each time the user presses **ENTER** to enter data. For example, assume form X is a repeating form that overlays itself:



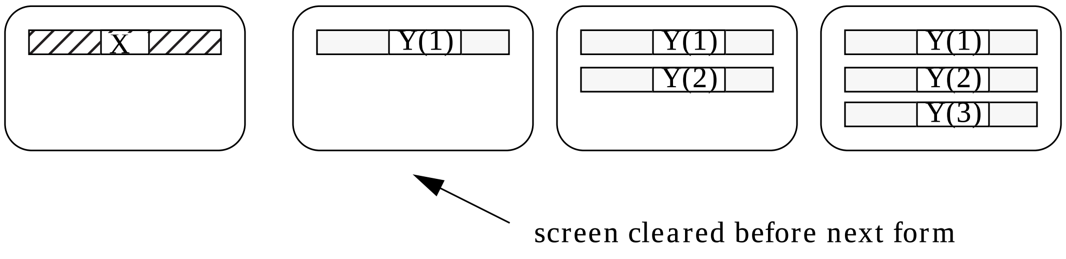
Note that this type of repeating form overlays itself even if there are other forms on the screen.

The Next Form Option

You specify the name of the next form to be displayed after the current form or keep the default of $HEAD. You may also specify whether the next form is to be appended to the current form (A); and, if appended, whether the current form is to remain frozen on the screen when the screen fills up (F). If the screen is to be cleared before the next form is displayed, keep the default of C.

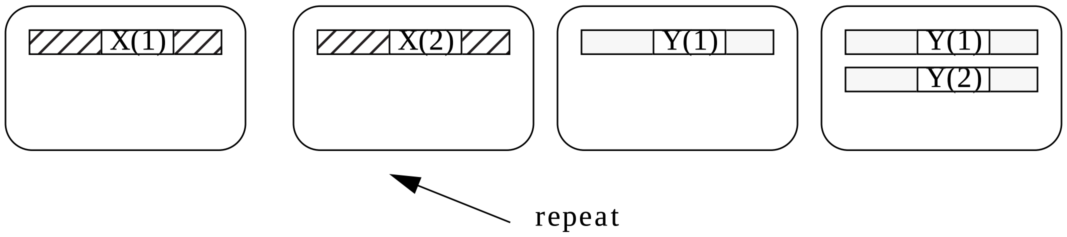
The following examples illustrate how freeze (F) and append (A) interact with the current form. Assume a current form X and a next form Y. The next form (Y) is defined on its own Form Menu as a repeating form appended to itself (repeat option = A for Append).

1.
```
Current Form X -- Repeat Option = N
                  Next Form Option = C
Next Form Y    -- Repeat Option = A
```



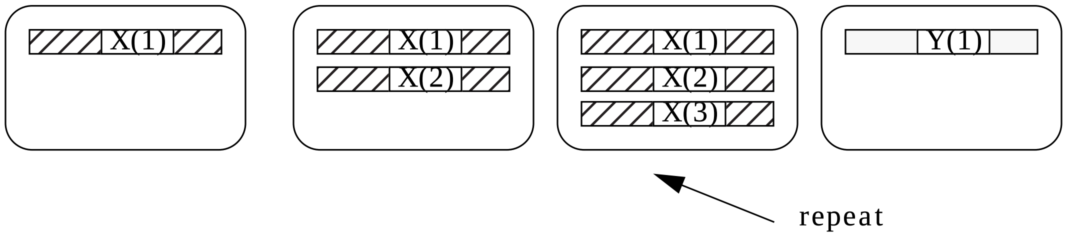
Form X is displayed, then after **ENTER**, the screen is cleared and form Y is displayed. After the next **ENTER**, Y is repeated below itself until the user presses NEXT or the application changes the repeat option.

2.
```
Current Form X -- Repeat Option = R
                   Next Form Option = C
Next Form Y    -- Repeat Option = A
```



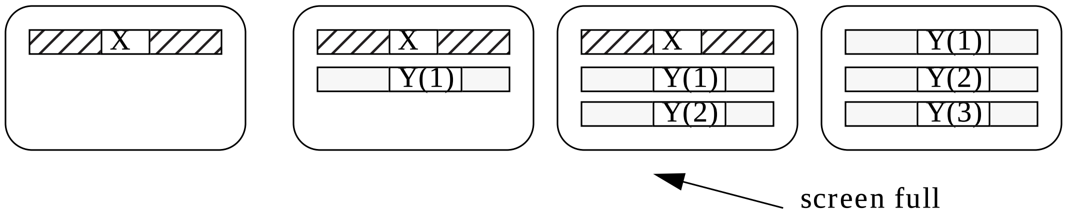
X is displayed until the repeat option is terminated, then Y is displayed and appended to itself until the repeat is terminated again.

3.
```
Current Form X -- Repeat Option = A
                   Next Form Option = C
Next Form Y    -- Repeat Option = A
```



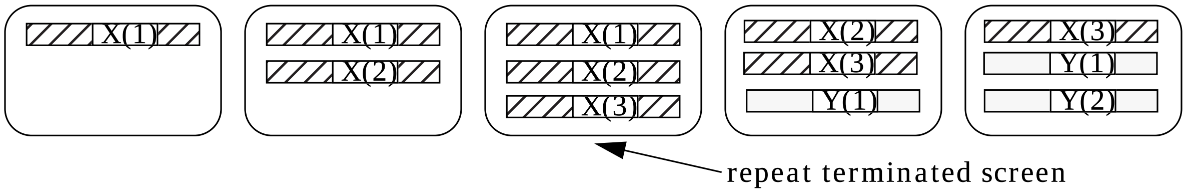
X is displayed and then appended to itself until the repeat option is terminated. Then the screen is cleared and Y is displayed.

4.
```
Current Form X -- Repeat Option = N
                   Next Form Option = A
Next Form Y    -- Repeat Option = A
```



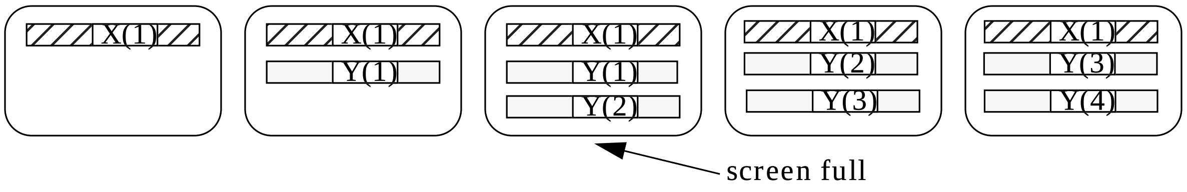
X remains on the screen while Y forms (in this example, two copies of Y) are appended to it until no room is left on the screen. When the third copy of Y is appended, form X (or part of it) is rolled off the screen and deleted to make room. However, if the next form Y(1) filled the screen below X, Y(1) would be rolled off and deleted before Y(2) is displayed.

```
5. Current Form X -- Repeat Option = A
                      Next Form Option = A
   Next Form Y    -- Repeat Option = A
```



Form X is repeated and appended until the repeat is terminated. When the first Y form is displayed, there is no more room for the first X and it is rolled off the top of the screen and deleted. As new Y forms are appended, the X forms continue to be rolled off and deleted.

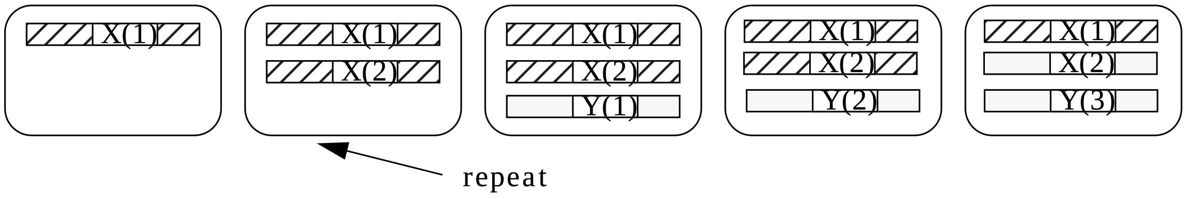
```
6. Current Form X -- Repeat Option = N
                      Next Form Option = F
   Next Form Y    -- Repeat Option = A
```



Form X remains frozen on the screen as form Y is appended to it. When the screen is filled, X remains frozen on the screen while the oldest Y form, Y(1) is rolled off to make room for Y(3). Note that if Y(l) fills the screen below X, it will be rolled off the screen and deleted when Y(2) is displayed.

Note that the F specification determines what happens when the screen is full, The A specification determines what happens when the next form is displayed. If an appended form does not fit on the screen with a frozen form, the freeze specification is cleared and the frozen form is rolled off the screen and deleted until the entire next appended form fits.

```
7. Current Form X -- Repeat Option = A
                      Next Form Option = F
   Next Form Y     -- Repeat Option = A
```



Form X is appended to itself. When the repeat is terminated, the first Y form is appended to the last X. When the next Y form is displayed, the first Y is rolled off the screen and deleted leaving the remaining X forms on the screen.

Note that if one frozen X forms should fill the screen, the first X is rolled off and deleted to make room for the latest X, and when the first Y form is appended, the X forms are no longer frozen.

Sample of Forms Design

Before running FORMSPEC to define your forms file, it is very helpful to decide on some of the basics of forms design, such as:

- Consider what kind of data the user will be entering — character, numeric, or date.
- Determine the position of the fields — what order, how many on a line, which need to be on the same line, the same form.
- Decide on which enhancements to use for each field.
- Take into account what type of terminal the users will have. Are there any special features? Will they be used?
- What native language will be used?
- Will the forms file be used with ENTRY or with an application? How will form sequencing be handled? Do function keys need to be defined? If so, which ones?

Not all of these questions need be answered prior to forms design, since FORMSPEC provides defaults for most form and field characteristics. Still, you may find it helpful to roughly sketch each form layout on paper, something along the lines of the example in Figure 3-11. The field name and length should be noted, and if a field has special (nondefault) characteristics, these too may be noted on your preliminary sketch, as was done in Figure 3-11. Preparing your forms layout in this way allows you to then sit down at the terminal and actually specify the complete forms file in a matter of minutes. You may find that completing the form layout and accompanying Field Menus is sufficient to define the characteristics of all fields on the form. In many cases, the default values supplied by FORMSPEC can be used, thereby reducing your actual input to a minimum. However, as you become familiar with the capabilities of VPLUS, including the processing specifications of FORMSPEC, described in Section 4, and the intrinsics available to applications, described in Section 6, you may find yourself taking advantage of the additional options as you finalize the design of your forms file.

The example in Figure 3-11. answers many of the questions listed above, both graphically and in accompanying notes. This forms design, when entered through the FORMSPEC menus, will generate a set of forms similar to those used as a data entry example with ENTRY in Section 2. (What data will your application need? How should it be laid out?) For forms sequencing, note that the first form is "frozen" on the screen, the second form is appended to the first and is repeated until the user presses NEXT to display the next and last form, TOTALS. (In this example, since ENTRY is used, the ENTRY function keys are used as well as the forms sequencing options of the Form Menu. Is this true for your design?) When TOTALS is displayed, all previous forms are cleared from the screen. Up until that point the first form is frozen on the screen. Should the second form be repeated so many times that there is no more room on the screen, its first appearance is rolled off while the first form remains on the screen. (How will you handle multiple forms?) These are some of the questions you should consider when designing your own forms file.

Figure 3-11. Sample Forms File Layout

FORMS FILE NAME = ORDENT

(No GLOBALS)

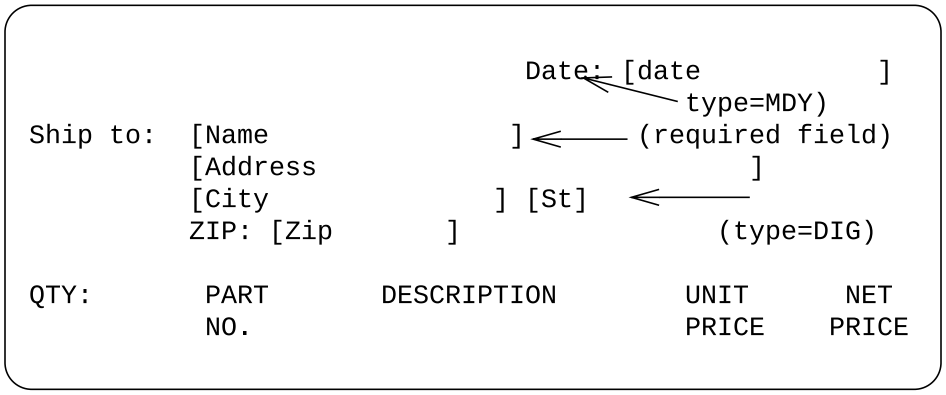
FORM #1 (head form)

Name: SHIPTO

Repeat = N <-- no repeat/append

freeze/append = F <-- leave form on screen with next form

Layout:



```
                                       Date: [date              ]
                                                  type=MDY)
Ship to:   [Name                    ] <------ (required field)
           [Address                                ]
           [City                 ] [St]  <-------
           ZIP: [Zip         ]                   (type=DIG)

QTY:         PART        DESCRIPTION           UNIT      NET
             NO.                               PRICE     PRICE
```

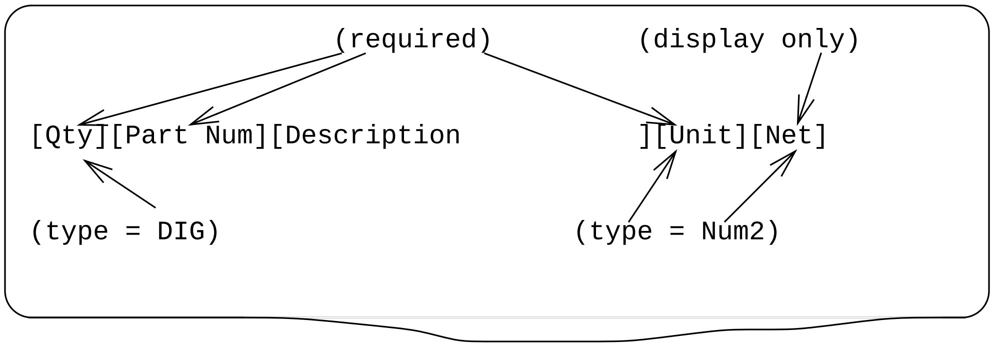
FORM #2
Name= ORDER

Repeat = A<-- Form is repeated and appended

freeze/append = C<-- When next form is requested
by user or program, current
form is cleared.

Next Form = TOTALS

Layout:



```
                    (required)          (display only)

[Qty][Part Num][Description                ][Unit][Net]

(type = DIG)                            (type = Num2)
```

FORMS FILE

The forms file consists of "global" specifications that apply to all forms in the file, followed by the individual forms specification. Within each form specification, the form is identified, and its form layout is defined. The form layout defines each field into which data can be entered. Each of these different types of specification is entered on a menu displayed by FORMSPEC. (Refer to Figure 3-12. for an illustration of a prototype forms file.)

Forms Modification

At any time during forms design, you can change any form or field currently specified in the forms file. The function keys or the Main Menu allow you to return to any existing form or field specification. You can then simply change the field or form and press **ENTER**. The new specifications override those previously entered. Refer to "Ease of Forms Design" earlier in this section.

Another method of modifying a form file is to use the delete option on Main Menu. This option allows you to delete an entire form or to delete fields global to all forms (save fields). When you delete a form, be sure to modify any other form that references the deleted form. For example, if FORM2 is given as the next form name for FORM I, and FORM2 is deleted, you must change the next form name specification on the menu for FORM1. Note that when you delete a parent form, all child forms of that form family will be deleted also. Refer to the "Main Menu" menu description and the "Form Families" discussion for more information.

Copying of Processing Specifications

The processing specification copy function allows processing specifications to be copied from one field to another field within and across forms files. Processing specifications are entered from the Field Menu of the FORMSPEC utility. This function is also invoked interactively from the Field Menu.

To use this function, enter one of the following commands in the first line of the processing specifications area in the Field Menu:

- `#COPYTO` *`newfilename`*
- `#COPTFROM` *`oldfilename`*

If processing specifications exist on the first line, press **Insert Line** before entering the command. After entering one of these commands, press **F3** to execute the command.

NOTE These commands are not case sensitive.

#COPYTO

When `#COPYTO` *`newfilename`* is executed, the processing specifications defined for the field are copied to the file specified by *newfilename*. As the template suggests, the file specified should be a new file. If there are no processing specifications to copy, the new file is not created. The command should then be deleted from the processing specifications area. If any error is encountered in executing the command, it is displayed in the upper area of the Field Menu screen.

#COPYFROM

When `#COPYFROM` *`oldfilename`* is executed, the processing specifications (if any) in the file specified by *`oldfilename`* are copied into the processing specifications area of the field. Press **Enter** after the command executes (successfully) to save the copied processing specifications for the field.

Since this command is used to retrieve the processing specifications, it cannot be used if specifications already exist for the field. If any error is encountered in executing the command, it is displayed in the upper area of the Field Menu screen.

The copy option on the Main Menu provides yet another method of modifying a form file. This useful option allows you to copy an entire form from the current forms file or from another forms file. The copied form is created by FORMSPEC as a new form that is an exact replica of the existing form. The form being copied must be given a name unique to the current forms file. The copied form can then be displayed and modified to suit your design.

The copy option does not copy save fields from one file to another; you must recreate the necessary save fields if you copy a form from a forms file other than the current file. Also, form family relationships are not maintained when forms are copied from one file to another. You may re-establish form family relationships using the relate option on the Main Menu. Refer to the "Main Menu" description for more information.

Most of the modifications can also be made using FORMSPEC in batch mode. (See Section 7 for more information.)

Error Messages

The following error messages may be displayed when executing the `#COPYTO` *newfilename* command:

- If the file specified by *newfilename* already exists, the following message is displayed and no copying is done:

  ```
  File already exists!
  ```

- If the syntax of the command is not proper, the following message is displayed:

  ```
  Check syntax: #COPYTO newfilename.
  ```

- If the file attributes check for a new file results in an error, the following message is displayed:

  ```
  New file check failed!
  ```

  The specified file should be a non-existent permanent file.

- If the file cannot be opened, the following message is displayed:

  ```
  Cannot create new file!
  ```

The following error messages may be displayed when executing the `#COPYFROM` *`oldfilename`* command:

- If *`oldfilename`* cannot be opened, the following message is displayed:

  ```
  Unable to open old file!
  ```

- If the syntax of the command is not proper, the following message is displayed:

  ```
  Check syntax: #COPYFROM oldfilename
  ```

- If processing specifications exist for the field, the following message is displayed:

  ```
  Cannot do a #COPYFROM if proc specs already esist.
  ```

When a new file is created with the #COPYTO command, it is created in the permanent file domain. Using this file in a subsequent #COPYFROM command does not purge this file. Purge this file manually if it is no longer required.

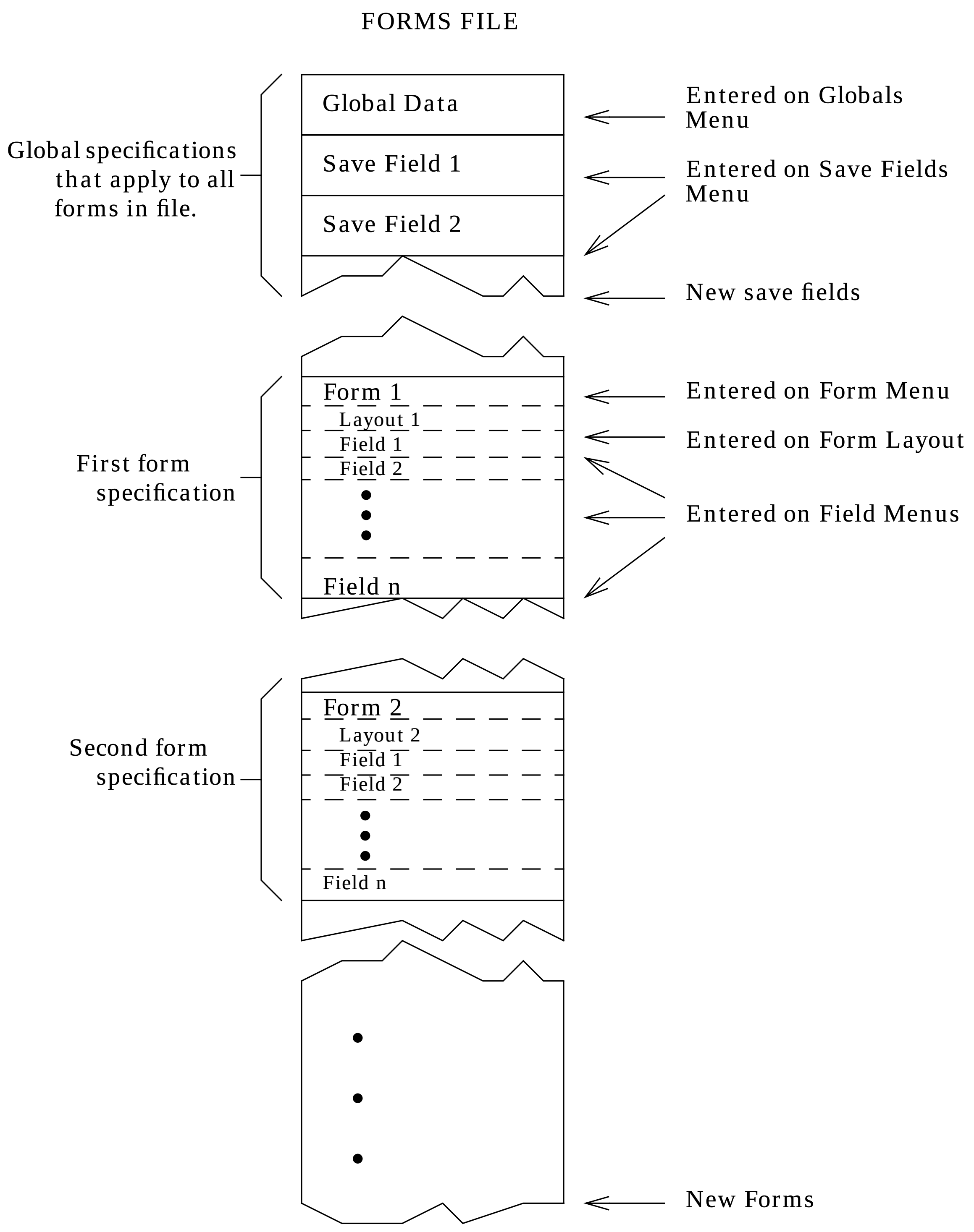
Forms File Size

FORMSPEC indicates what percentage of the forms file is full whenever the Main Menu is displayed from the Forms File Menu. The percentage is determined by the formula:

```
file EOF / file limit
```

This message is designed to give the user an idea of the space utilized in the forms file. Since a single forms file can contain an essentially unlimited number of forms (subject to system constraints), the forms file may reach its end-of-file (default=4000 records). When the percentage indicated on the Main Menu grows large, you should increase the size of your forms file, as described below.

Figure 3-12. Forms File Prototype



Expand Forms File

When you require the forms file to be larger than the default limit of 4000 records, you may override the FORMSPEC default size to create a new forms file using the MPE :FILE command. The :FILE command must be issued prior to the :RUN command to initiate FORMSPEC. Another method is to use the MPE :BUILD command to create a larger file and then use FCOPY to copy an existing forms file into the larger file.

To use the :FILE command to create a forms file named ORDFORM, issue the file equation:

```
:FILE ORDFORM;DISC=6000
                   ^
           desired disc space, in number of records
```

Run FORMSPEC and specify ORDFORM as the forms file. You now have an empty file with room for 6000 records, into which you may add or copy forms using the FORMSPEC Main Menu options. Note that the FORMSPEC copy Main Menu option will not copy save fields from other forms files or maintain form family relationships across forms files. You may re-establish form family relationships using the FORMSPEC relate Main Menu option. Refer to the "Main Menu" description for more information.

To use the :BUILD command to expand your forms file, ORDFORM, give the following command:

```
:BUILD ORDFORM2;DISC=6000;REC=128,1,F,ASCII;CODE=VFORM

             desired disc space
```

Run FCOPY to copy the contents of ORDFORM into the larger file:

```
:RUN FCOPY.PUB.SYS
>FROM=ORDFORM;TO=ORDFORM2

EOF FOUND IN FROMFILE AFTER RECORD 1015   <- FCOPY response

1016 RECORDS PROCESSED *** 0 ERRORS
```

Compiling the Forms File

The forms defined through FORMSPEC are written to a forms file. The forms are initially stored as a "source" version. This source is modified if you change the forms file, but it must be compiled before it can be executed by an application, The compiled version, on the other hand, can be executed but not modified. The source version of the forms file is kept along with the compiled version for purposes of display and modification. Once modified, the source version must be recompiled before it can be executed. However, unless a global attribute of the forms file is modified, only those forms which were changed are actually recompiled. The global attributes include any save fields or the fields on the Globals Menu or the Terminal/Language Selection Menu.

Note that the source version is the sequence of specification records entered on the menus. The compiled version is the sequence of forms displayed to the user.

When your forms design is complete, you can compile the forms file to a "fast forms file". A fast forms file is similar to the forms file except that it is created with the smallest record size that can hold the largest form in the file. Also, it contains only the information that VPLUS needs at run-time, so it has fewer records. Because the record size and the number of records are minimal, such a forms file can improve performance at run-time. A fast forms file can only be executed; it cannot be modified. You may however always modify the source forms file and then recompile it to a fast forms file.

The forms file can also be compiled using FORMSPEC in batch mode. (See Section 7 for more information.)

Renumbering a Form with Interactive FORMSPEC

FORMSPEC provides an option to renumber forms interactively. A new command has been added to the Interactive FORMSPEC to renumber forms. This command is invoked by selecting option “N” on the Main Menu. The form to be renumbered should be specified in the field next to the keyword “N”. (This field is the same as where you would specify a form when also selecting the “D” option.) This option is equivalent to the RENUMBER command in batch mode FORMSPEC.

Listing Forms

Figure 3-13. illustrates the listing for a forms file. Note that the listing includes the current status of the forms file, including when it was last modified and compiled, and the number of two-byte words of stack space needed by VPLUS when accessing this forms file at execution time.

Forms can also be printed using FORMSPEC in batch mode. (See "Using FORMSPEC in Batch Mode", Section 7 for more information.) Consult Appendix E, for information regarding the file equations which VPLUS uses when listing forms files.

Figure 3-13. Forms File Listing

```
*********************************************************************
*                      FORMSPEC Version v.uu.ff                     *
*                     Wed, APR 16. 1986,   3:34 PM                  *
*                                                                   *
*                      ORDFORM1.PUB.ACCTG                           *
*                                                                       *
*************************************************************************
Forms File Status
   Modified: WED, APR 16, 1986, 3:25
   Compiled: WED, APR 16, 1986, 3:25
  Requires 5605 bytes of data space in addition to the applciation
   defined Comarea. (Subtract 1600 bytes if only using fast forms files.)
                  Head form: ORDER_HEAD
Default Display Enhancement: HI                                           Global
          Error Enhancement: IU                                           Information
        Window Display Line: 1
             Window Enhancement: HI

THERE ARE 4 SAVE FIELDS IN THIS FORMS FILE:
   Save Field: SORDNUM          Length: 6         Date Type: DIG
      Init Value: 100001
   Save Field: SNAME            Length: 42        Data Type: CHAR         Save Field
      Init Value:                                                         Information
   Save Field: STOTNET          Length: 10        Data Type: NUM2
      Init Value: 0
   Save Field: LINE_COUNT         Length: 2         Data Type: DIG

There are 3 forms in this forms file:
             Field Counts      Largest   Number of
             Total/Display     Field     Lines in
Form Name       Only             Number     Screen        Next Form       Summary of Forms
ORDER_HEAD      8   / 2         8        16            ORDER_LINE         in Forms File
ORDER_LINE      5   / 7        13         1            ORDER_TOTALS
ORDER_TOTALS     11  / 7         11        19            $HEAD

FORMSPEC VERSION v.uu.ff              WED, APR 16, 1986, 3:34 PM          Example of a
FORMS FILE: ORDFORM1. PUB.ACCTG                              page 2       Page Heading

Form: ORDER_HEAD
   Repeat: Option: N                                                      Example of a
   Next Form Option: F                                                    Form Menu
   Next Form: ORDER_LINE
   Reproduced from:

Comments: EXECUTE THIS FORM FIRST
******* ******* ******* ******* ******* ******* ******* *******

                    ****ABC MANUFACTURING****

Order Number: [ordnum]                    Date: [ordate          ]        Example of a
                                                                          Form Layout
SHIP TO:
    Name [name                               ]
Address  [address                            ]
   City  [city                        ] State [st]
   Zip  [zip      ] Telephone [phone          ]
```

Figure 3-13. Forms File Listing (Cont)

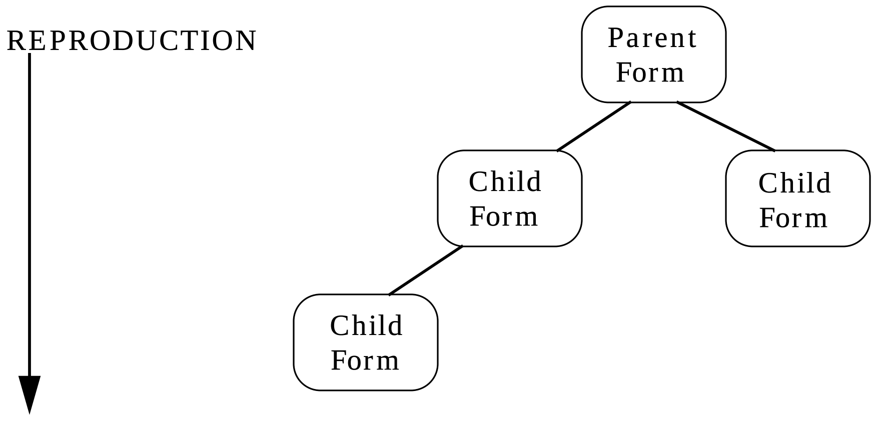
```
QTY    PART NO.          DESCRIPTION                UNIT        TOTAL
                                                    PRICE       PRICE
******* ******* ******* ******* ******* ******* ******* *******

Field: ordnum
   Num:8    Len:6    Name:ORDNUM      Enh: HI   FType: D   DType: DIG
   Init Value:
*** PROCESSING SPECIFICATIONS ***

INIT
   SET TO SORDNUM                    \Mover order number to this field
   SET STOTNET TO 0                  \Initialize totals save field
   SET LINE_COUNT TO 0                  \Initialize counter for order lines


Field: ordate
   Num:1    Len:18   Name:ORDATE      Enh: HI   FType: D    DType: MDY
   Init Value:
*** PROCESSING SPECIFICATIONS ***
INIT
   SET to $TODAY                        \Assign today's date to field
```

Example of
Field Menus

FORM FAMILIES

A family of forms is a collection of forms that share a common form layout but may have different field attributes, processing specifications, or form sequencing options. Use of form families may improve VPLUS performance in some transaction processing applications; since only the internal form characteristics are different among members of a family, you do not have to wait for VPLUS to repaint the screen when you change to another form in the same family. Note that the security attribute is not updated when changing from one form in a family to another in the same family. Figure 3-14. illustrates some form family concepts.

Figure 3-14. Form Family Relationship



`Parent Form`	Also called father or root form. A parent form is a standard VPLUS form created in FORMSPEC either by designing it or by copying it from another form. The form layout of the parent form determines the layout of all child forms that are reproduced from it. Changing the layout of the parent form causes the layouts of the child forms to change.
`Child Form`	Reproduced from the parent form, the form layout of the child form may not be modified. You may change form sequencing options, field attributes and processing specifications. As Figure 3-14. indicates, you may reproduce a child form from another child form, although VPLUS still considers the original root form to be the parent of both forms.
`Reproduce`	Generate a related form by entering the name of an existing form in the "Reproduced from" field of the Form Menu.

Creating Child Forms

You create child forms by following these steps:

1. Select **A** for "Add a form" at the Main Menu. Press **ENTER**.
2. When the Form Menu comes onto the screen, specify the name of the child form you wish to create, and, in the "Reproduced from" field, specify the name of the form from

which you wish to generate the child form. You may change any of the other Form Menu fields. Press **ENTER**.

3. The layout for the form you named in step 2 is displayed. Then press **NEXT**. Since the form layout must be the same for parent and children, pressing **ENTER** causes the error message "Cannot change screen, the form is a family member." to appear in the window.
4. You may change any of the attributes or processing specifications on the Field Menus (except length and field number) to adjust the internal characteristics of the child form to your needs. A child form continues to have all the characteristics of the parent if you do not modify the child's Field Menus.

To modify the form layout of a form family, you need to change only the layout of the parent form, and recompile the forms file. This changes the layouts of the child forms automatically. Deleting the parent form deletes the entire form family.

Careful naming of forms and the use of the optional comment field of the Form Menu makes relationships between forms of the same family more apparent. To make it easier to distinguish between forms of the same family in ENTRY or in other applications, you may wish to add a field reserved for the form name to the screen definition of the parent form, and add processing specifications which set the value of that field to the name of the current form. Refer to the SET command in Section 4.

Although family relationships are not maintained after a copy operation, they can be re-established using the **R** option "Relate child form to parent form" at the Main Menu. In order to use this method of creating family relationships, certain criteria must be met. The forms must already exist and their form layouts must be identical. The forms must have identical field number sequences. The forms must be distinct forms; that is, one form cannot be both the parent form and the child form. Furthermore, the child form cannot already be a child form or a parent form. Thus, it is also possible to relate forms which were never previously related.

Users should note that if a new form is to be reproduced from a form which is already a child, then its parent will be designated as the actual parent. For example, suppose that an existing forms file contains three forms named `PARENT`, `CHILD` and `NEWCHILD`, where `PARENT` and `CHILD` already maintain a relationship. At the Main Menu, it is indicated that `NEWCHILD` should become a child form of `CHILD`. If the forms meet the above criteria, FORMSPEC actually creates the relationship with `PARENT` as the parent form and `NEWCHILD` as its child. The form named `CHILD` continues to be a child form of `PARENT`.

Forms can also be related using the RELATE command from FORMSPEC in batch mode. (For further information see "Using FORMSPEC in Batch Mode", Section 7.)

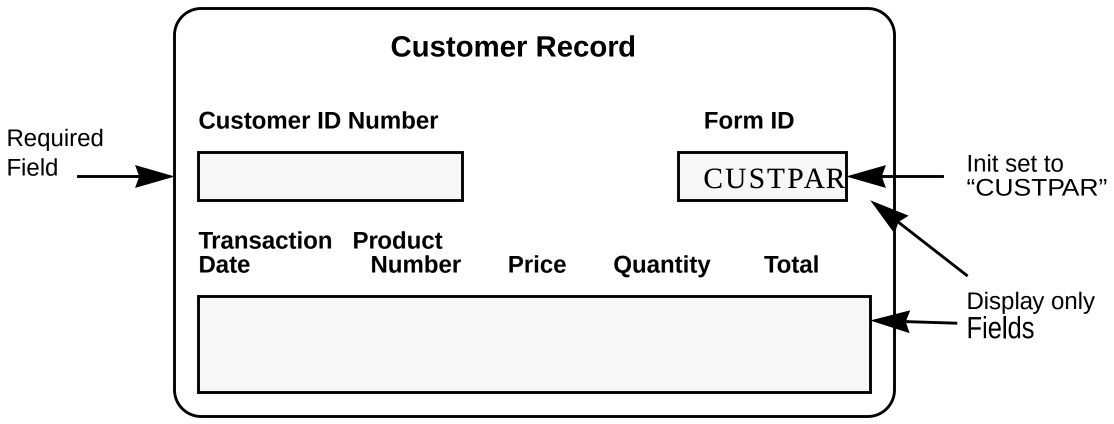
Form Family Example

Suppose you have a VPLUS application which, after accepting a customer's identification number, retrieves items of pertinent customer information from a data base and displays them on a form. It would be convenient to have two forms, one on which the customer ID is required, and the other on which the user views or alters the customer data. If you don't use the form family feature, the user who is alternating between entering customer ID's and inspecting or modifying data constantly has to wait for the form to be repainted each time the user needs information on another customer.

The solution to this problem of long user waiting time is to create two forms with the same form layout but different field attributes.

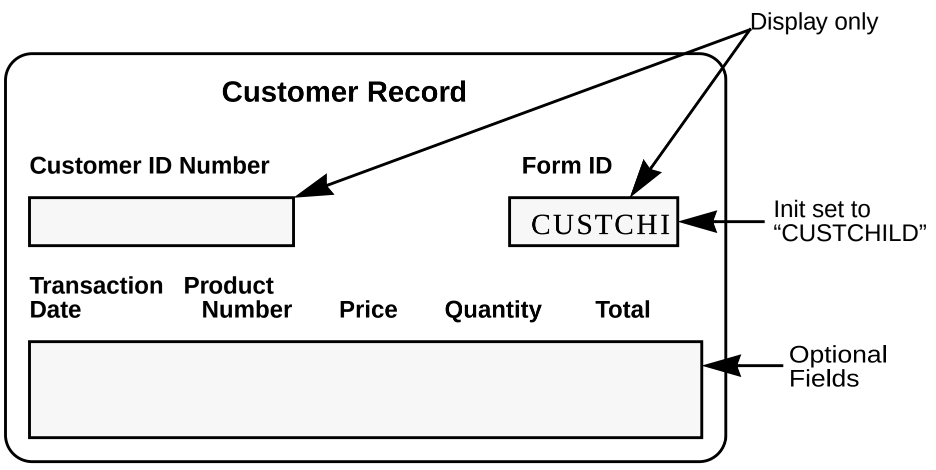
Suppose the root or parent form of this example is named CUSTPAR, and that its form layout is as shown in Figure 3-15. below. All of the fields in this form are display only, except for the Customer ID Number field, which is required.

Figure 3-15. Parent Form



Create a child form, CUSTCHILD, by specifying CUSTPAR in the "Reproduced from" field of the Form Menu. Press **NEXT** until the Field Menu for CUSTID appears and make it a display only field. Make all other data fields in the form optional. Compile the forms file.

Figure 3-16. Child Form



Now when the user types the customer ID and presses **ENTER**, the application retrieves the related information from the database and displays it in the form immediately. Once the form is on the screen, it appears to remain there as long as the user uses the application, rather than being repainted each time for every customer about whom the user calls up data.

Fields

`Forms File Name` Enter the name of the forms file in this field. It may consist of up to 36 characters, as shown in Table 3-10.

NOTE In order to maintain (and browse or list) a forms file, the user must have exclusive write access to the file.

FORMSPEC accepts only forms files that are MPE files with the file code of VFORM. If you have a forms file that is a KSAM file, created by versions of VPLUS prior to A.01.01, refer to Appendix H for instructions on converting KSAM files to MPE files.

Table 3-10. Maximum Number of Characters in Fields

Element of File Name	Maximum Number of Characters
File Name	8
Group Name	8
Account Name	8
Lockword	8
2 periods, 1 slash	3
Terminator (blank or special character)	1
TOTAL	36

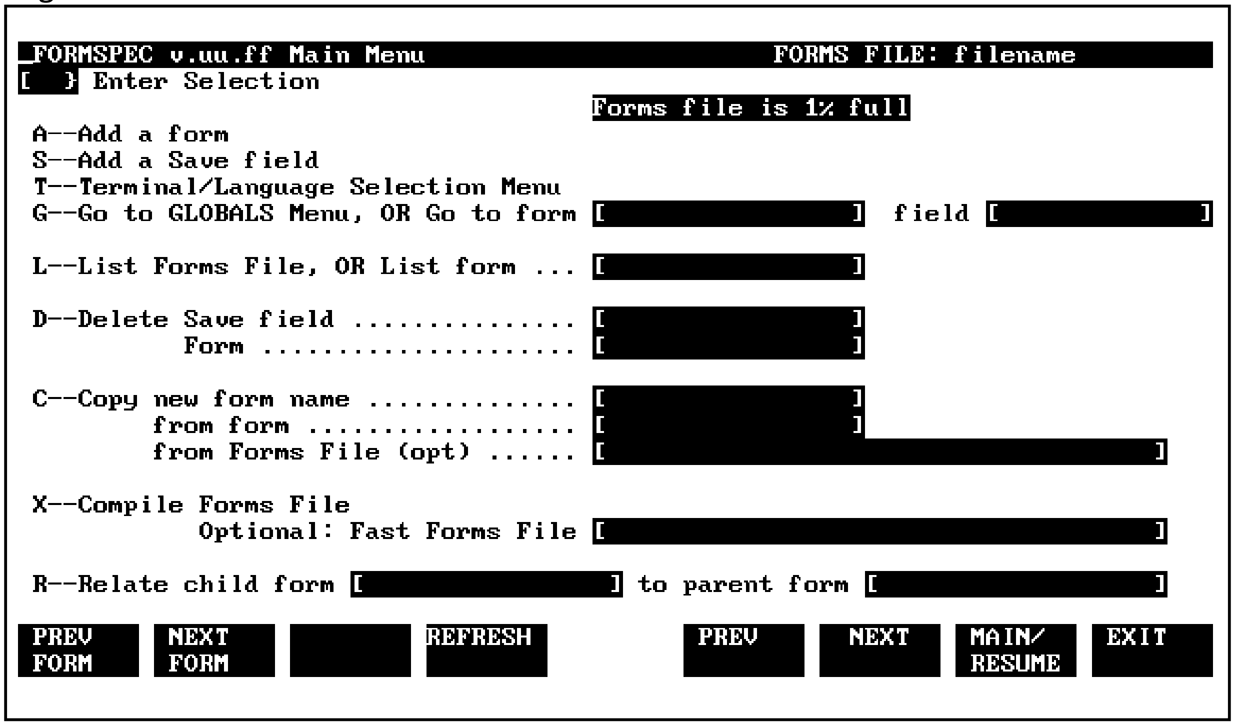
NOTE If your file has a lockword, you *must* enter the lockword along with the file name, as *filename/lockword*. If you do not enter a lockword with the file name and one is required, your terminal hangs. This is because the MPE prompt requesting the lockword is in character mode and cannot be received while your terminal is in block mode. You can recover from this error by doing a hard reset followed by pressing **RETURN** at least four times, then pressing EXIT. How you perform a hard reset depends on what type of terminal you are using; consult your terminal manual for instructions.

MAIN MENU

The Main Menu, as shown in Figure 3-17., is the main control menu for all FORMSPEC operations. If the forms file is new, you usually select **A** for Add a form. FORMSPEC then displays the menus that allow you to define your forms file, as described in "Ease of Forms Design". If the forms file already exists, you may enter any selection depending on what you want to do, such as add new forms to the file, select a particular menu in order to change it, delete a form, and so forth. Most of the options available on the Main Menu are also available using FORMSPEC in batch mode. (See Section 7 for more information.)

After modifying the forms file, you must compile it before it can be executed with the modifications. Only those forms of the existing file which have been changed are actually recompiled, unless a global attribute of the forms file, such as a save field, has been modified.

Figure 3-17. Main Menu



```
_FORMSPEC v.uu.ff Main Menu                           FORMS FILE: filename
[   ] Enter Selection
                                    Forms file is 1% full
 A--Add a form
 S--Add a Save field
 T--Terminal/Language Selection Menu
 G--Go to GLOBALS Menu, OR Go to form [                 ]  field [              ]

 L--List Forms File, OR List form ... [                 ]

 D--Delete Save field ............... [                 ]
           Form ..................... [                 ]

 C--Copy new form name .............. [                 ]
         from form .................. [                 ]
         from Forms File (opt) ...... [                                     ]

 X--Compile Forms File
            Optional: Fast Forms File [                                     ]

 R--Relate child form [                 ] to parent form [                   ]

PREV     NEXT              REFRESH          PREV     NEXT     MAIN/    EXIT
FORM     FORM                                                 RESUME
```

Fields

A--Add a Form

To add a form, simply type **A** in the selection box and press **ENTER**. In response, FORMSPEC displays a Form Menu, as shown in Figure 3-18., so you can define the form. For details of form definition using this menu, see the description below under "Form Menu".

S--Add a Save field

If you specify **S** in the selection box and press **ENTER**, a blank Save Field

Menu is displayed. This menu, as shown in Figure 3-24., allows you to specify a save field description. If you want to modify a save field, you use **PREV** or **NEXT** to locate the particular Save Field Menu specification. For more information, refer to the "Save Fields Menu" description.

`T--Terminal/Language Selection Menu`

If you specify **T** in the selection box and press **ENTER**, the Terminal/Language Selection Menu is displayed. This menu, as shown in Figure 3-31., allows you to specify the terminal or set of terminals as well as the native language (if utilizing Native Language Support) with which the forms file will be used. For more information regarding Native Language Support, see Section 8. (Refer to "Terminal/Language Selection Menu" below for detailed information.)

`G--Go to Globals Menu`

Type **G** in the selection box, but do not specify a form or field name. Press **ENTER**. The Globals Menu, as shown in Figure 3-26., is displayed so you can make changes to the global characteristics of the forms file. It provides access to the Data Type Conversions Menu. (Refer to "Globals Menu" for detailed information.)

`G--OR Go to form`

To display the Form Menu for a specific form, type **G** in the selection box, specify the form name, and press **ENTER**. (You can also locate the Forms Menu with `PREV FORM` or `NEXT FORM`. When the Form Menu is displayed, you can then change any specification on the menu. (Refer to "Form Menu" below for a discussion of the form description.)

`G--OR Go to field`

The Field Menu for a specific field can be located by typing **G** in the selection boxes then specifying both the form and field name, and pressing **ENTER**.

You can display or change field descriptions either on the Field Menu or directly during the form layout. The field attributes (except for length and number) and processing specifications can be entered and changed directly on the Field Menu. Any new fields are added, existing fields deleted, or the number or length of fields are changed by changing the form layout. (For details, refer to the "Ease of Forms Design" discussion earlier.)

`L--List Forms File OR List form`

Type an **L** in the selection box, specify the form name and press **ENTER** to print a description of any form in your forms file. If you want to list all the forms in the file with a description of the file in general, simply type **L** but do not specify a form name. Press **ENTER**. Refer to the example listing in Figure 3-13. The listings are printed on the standard list device (`;DEV=LP`, usually, the line printer) if there is no overriding MPE `:FILE` command. Consult Appendix E for information on the file equations and the JCW that VPLUS uses when listing forms files.

`D--Delete Save Field`

To delete a save field, specify **D** in the Main Menu selection box, specify the save field name, and press **ENTER**. The field will be deleted. You should be sure there are no references to the deleted field in the processing specifications of any other field (refer to Section 4).

`D--Delete Form`

You can delete an entire form from the forms file by typing **D** in the selection box on the Main Menu, specifying the form name, and pressing **ENTER**. If the form has an ARB associated with it, the ARB is deleted at the same time.

`C--Copy new form name from form from Forms file (opt)`

To copy a form, type **C** in the selection box. Then specify the name you want to give the new form, the name of the existing form to be copied, and, if the form to be copied is not in the current file, the name of the file containing this form. Press **ENTER**. If the form has an ARB associated with it, the ARB will also be copied.

`X--Compile Forms File`

You can compile the current forms file by typing **X** in the selection box on the Main Menu and pressing **ENTER**.

If the forms file has already been compiled, only new and modified information is recompiled unless a global attribute such as a save field, an item on the Terminal/Language Selection Menu, or an item on the Globals Menu has been modified. The new compiled version replaces the previous version. There is never more than one compiled version of a forms file at a time. You may however modify the source version without affecting the compiled version. For example, if the compiled version is being used for data entry, it is not altered in any way as you modify the source version. When the source is completely modified, you can compile it and use the newly compiled version for data entry. In that case, the previous compiled version is lost.

`X--Compile Optional: Fast Forms File`

To compile to a fast forms file, type **X** in the Main Menu selection box exactly as you would for any compilation (or recompilation), but also supply the fast forms file name and press **ENTER**.

`R--Relate child form to parent form`

Form family relationships can be created by typing an **R** in the Main Menu selection box, and the names of the two forms which are to be "related" in the Relate fields and pressing **ENTER**. The forms must already exist and their form layouts must be identical. The forms must be distinct forms; that is, one form cannot be both the parent form and the child form. Furthermore, the child form cannot already be a child form or a parent form. Thus, it is also possible to relate forms which were never previously related. Family forms and their characteristics are fully discussed under

"Form Families" earlier in this section.

`B--Go to ARB Menu`

To create, modify or delete an application-ready buffer (ARB), type **B** in the Main Menu selection box, and the name of the associated form in the ARB Menu field, and press **ENTER**. The form must already exist in the forms file and the data type conversion record must have been set up using the Globals menu. Construction of an ARB is discussed in detail later in this section.

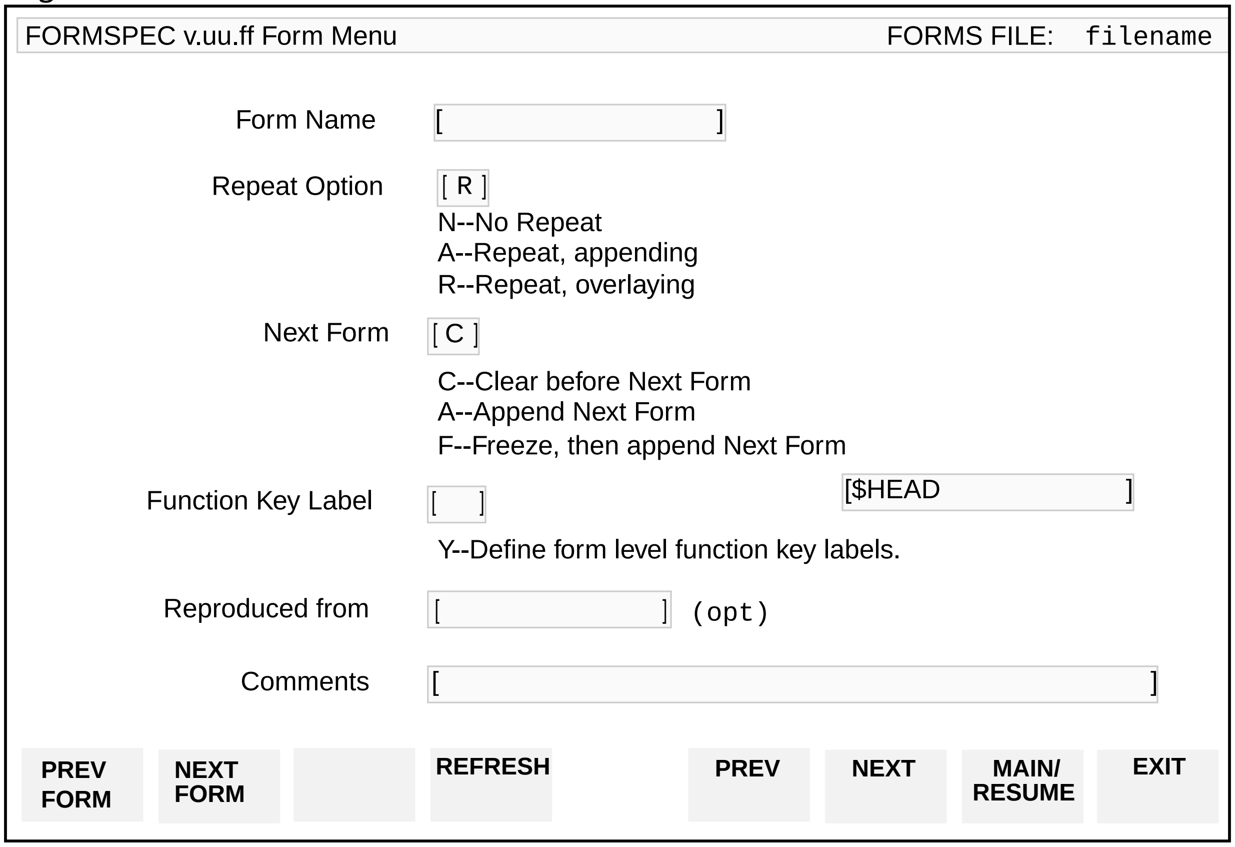
FORM MENU

The name and characteristics of each form are specified on the Form Menu. As soon as the Menu specifications are entered, FORMSPEC displays a blank screen on which you design the layout of the form. This form layout phase establishes the field names and their characteristics. Based on the form layout, FORMSPEC then displays a Field Menu for each field established on the form.

The Form Menu is displayed once for each form you define. When you have no more forms to define, press **MAIN/RESUME** to display the Main Menu in order to compile the forms file. The Form Menu is illustrated in Figure 3-18.

FORMSPEC stores forms in alphabetic order. When new forms are added to the forms file, they are inserted in the USASCII collating sequence. The Globals Menu (see description later in this section) allows you to specify which form is to be executed first. This form can be referenced by the system-defined value $HEAD. If you do not specify the head form name, $HEAD defaults to the first form in the file. Adding a new form may change the value of this default.

Figure 3-18. Form Menu


FORMSPEC v.uu.ff Form Menu FORMS FILE: filename

Form Name [ ]

Repeat Option [R]
N--No Repeat
A--Repeat, appending
R--Repeat, overlaying

Next Form [C]
C--Clear before Next Form
A--Append Next Form
F--Freeze, then append Next Form [$HEAD]

Function Key Label [ ]
Y--Define form level function key labels.

Reproduced from [ ] (opt)

Comments [ ]

You must specify at least a form name on the Form Menu. Default values are provided for the remaining specifications. If these defaults are not what you want, you can overwrite them with other values. If satisfied with the defaults, press **ENTER** to go directly to form layout.

Fields

Form Name

Enter a name to identify the form. The name can be from 1 to 15 characters, either letters of the USASCII alphabet (A-Z), digits (0-9), or the underline (). The first character must be alphabetic. The name can be entered using uppercase or lowercase letters, but all lowercase letters are shifted to uppercase by FORMSPEC. Thus, if you enter the name Form2, FORMSPEC changes it to FORM2.

The form name must be unique within the forms file, and it cannot be one of the reserved words listed earlier in Table 3-2. Note that the form name can be changed by replacing the name in the Form Name field. If you change a form name, you should make sure that references to this form as "Next Form" in other Form Menus are also changed.

Repeat Option

When the forms in the forms file are displayed, a form may be repeated and appended to itself (A); it may be repeated overlaying the previous display of the same form (R); or it can be a non-repeating form that is displayed on the screen once (N).

Either repeat option (A or R) causes the form to be repeated until the application changes the repeat option, or, in ENTRY, the user presses **NEXT** to request the next different form.

Default No repeat (N).

NOTE Both Repeat Option and Next Form can be changed dynamically within the application (refer to Section 6) as well as through processing specifications (refer to Section 4).

Next Form

You must enter the name of the next form to be displayed after the current form. You may also specify whether the next form is to be appended to the current form, A; and, if appended, whether the current form is to remain on the screen when the screen fills up, F. If the screen is to be cleared before the next form is displayed, leave the default value, C.

Default Clear (C).

`Next Form Name`	The next form name is entered as a standard form name or you can enter one of the following system-defined values:
`$RETURN`	Display the previous (but different) form.
`$HEAD`	Display the first form (either the first in the file or as defined on the Globals menu).
`$END`	Terminate forms display with this form.
`$REFRESH`	Display current form cleared to initial values as the next form.
	Note that each of these values has meaning only when the forms file is executed. Any value entered for Next Form Name can be changed through the `CHANGE NFORM` processing specification described in Section 4.
	Default `$HEAD`
`Freeze/Append Option`	This option allows you to specify how the next form will interact with the current form. The next form can be appended to the current form, and if so, the current form can be frozen on the screen when subsequent forms are displayed. Note that this option can also be changed by the `CHANGE NFORM` field processing specification described in Section 4.
	Default Clear screen before displaying next form (C).
`Function Key Labels`	To define or modify form level function key labels, enter a Y in this box to obtain the Form Function Key Labels Menu. After function key labels have been defined, FORMSPEC will place an X in the Function Key Label field. The default function key labels can be reset by entering an N. If you specify a Y in this field, the Form Function Key Labels Menu is displayed, as shown in Figure 3-19. The data capture devices do not support function key labeling.
`Reproduced from (opt)`	This optional field is used in the creation of form families, a feature discussed earlier in this section.
`Comments`	You can enter a comment to help document the form. The comment is displayed with the source file, but is not included in the displayed form at execution time. A comment can be up to 50 characters long.

FORM FUNCTION KEY LABELS MENU

This menu is used to specify function key labels that will be displayed along with the form. Labels you specify replace original labels provided by VPLUS. Each label consists of two lines of eight characters each, To specify a label, enter the first line in the first field and the second line of the label in the second field in the menu. Both lines can be specified on this menu. The color pairs fields allow you to specify a number, (`1 - 8`) to indicate a color for each function key. If color pairs are not specified, VPLUS supplies color pair 3 as the default. Functions keys are always shown in the inverse of the color pair supplied. That is, the foreground and background colors are exchanged.

Refer to Table 3-3. for the correct color to number mapping.

For example, to change the label for Key 1 to read:

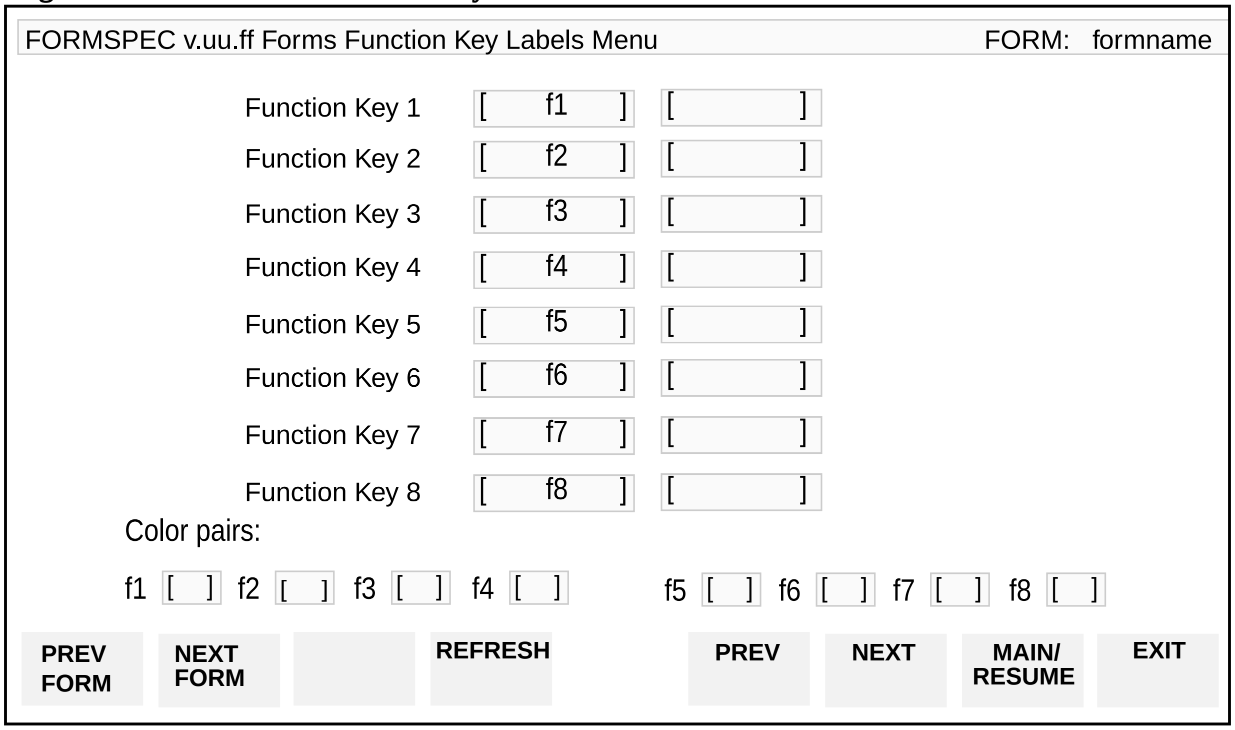
```
ORDER
NUMBER
```

Change the field for Key 1 as shown:

```
Function Key 1   [ORDER    ]  [NUMBER   ]
```

If the labels are to be Local, they will be displayed only while that form is on the screen. If the next form does not have Local labels, the file's Global labels will be displayed. In the case of a frozen form with another form appended to it, the labels displayed will be those for the appended form. If the appended form has no Local labels, the file's Global labels will be used. The function key labels can be updated using FORMSPEC in batch mode. (See Section 7 for more information.)

Figure 3-19. Form Function Key Labels Menu



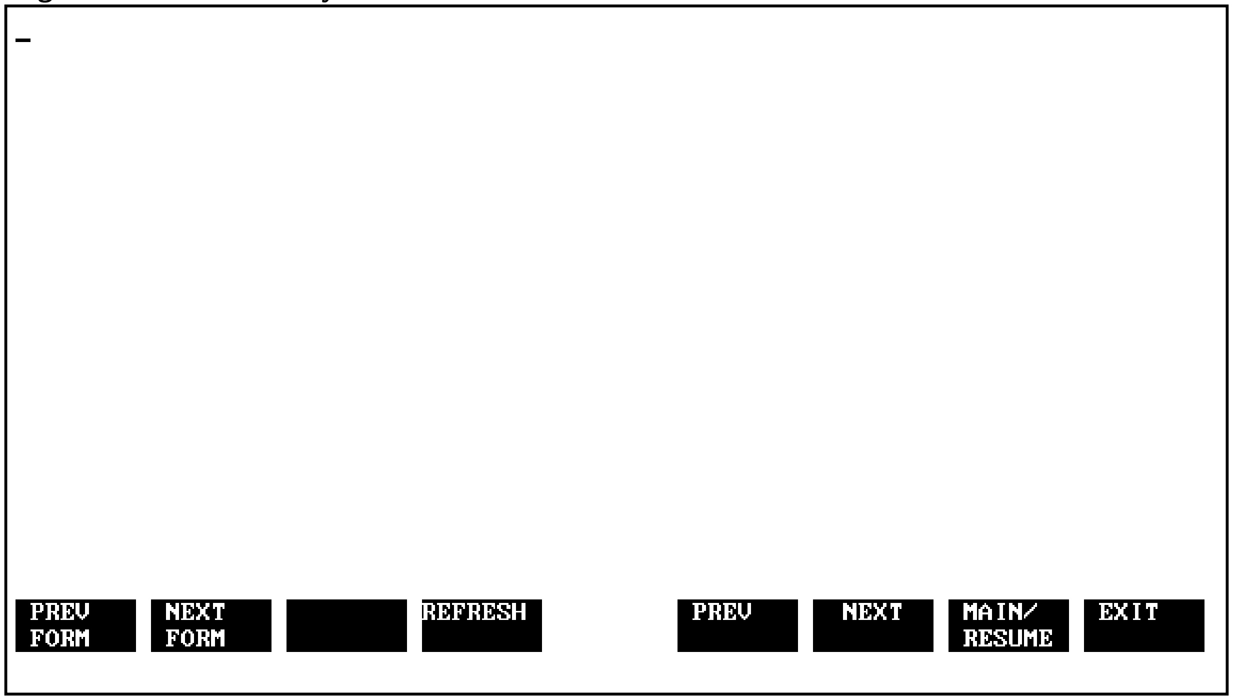
```
FORMSPEC v.uu.ff Forms Function Key Labels Menu                    FORM:   formname

                 Function Key 1      [   f1   ]  [        ]
                 Function Key 2      [   f2   ]  [        ]
                 Function Key 3      [   f3   ]  [        ]
                 Function Key 4      [   f4   ]  [        ]
                 Function Key 5      [   f5   ]  [        ]
                 Function Key 6      [   f6   ]  [        ]
                 Function Key 7      [   f7   ]  [        ]
                 Function Key 8      [   f8   ]  [        ]
        Color pairs:

        f1 [  ]  f2 [  ]  f3 [  ]  f4 [  ]      f5 [  ]  f6 [  ]  f7 [  ]  f8 [  ]

 PREV     NEXT              REFRESH       PREV     NEXT     MAIN/     EXIT
 FORM     FORM                                              RESUME
```

FORM LAYOUT

Each form is associated with a picture that is displayed when the form is requested during execution. This picture is called the "form". You design the form on a blank display screen, as shown in Figure 3-20., using all the terminal capabilities associated with block mode.

Once you have entered your headings and other text, and delimited the data fields, the form is basically designed. Each field has default field attributes assigned to it by FORMSPEC, so that unless you want to specify different attribute or processing specifications for the field, form design is complete.

Figure 3-20. Form Layout Menu



Note that you can use the terminal escape sequences or function keys to enhance the text. If you display another set of function keys (such as with **AIDS** or **MODES**) while designing the screen, you must restore the function keys to their FORMSPEC-defined values (such as with **USERKEYS**) before continuing forms design on the next menu.

Note that if you are designing a form to run on the HP 264X terminals, you must not use column 79 if the form is to be part of a form family nor use column 80 if the form is to be either appended or part of a form family. A multiline field must go all the way through column 80 in order to continue on the next line.

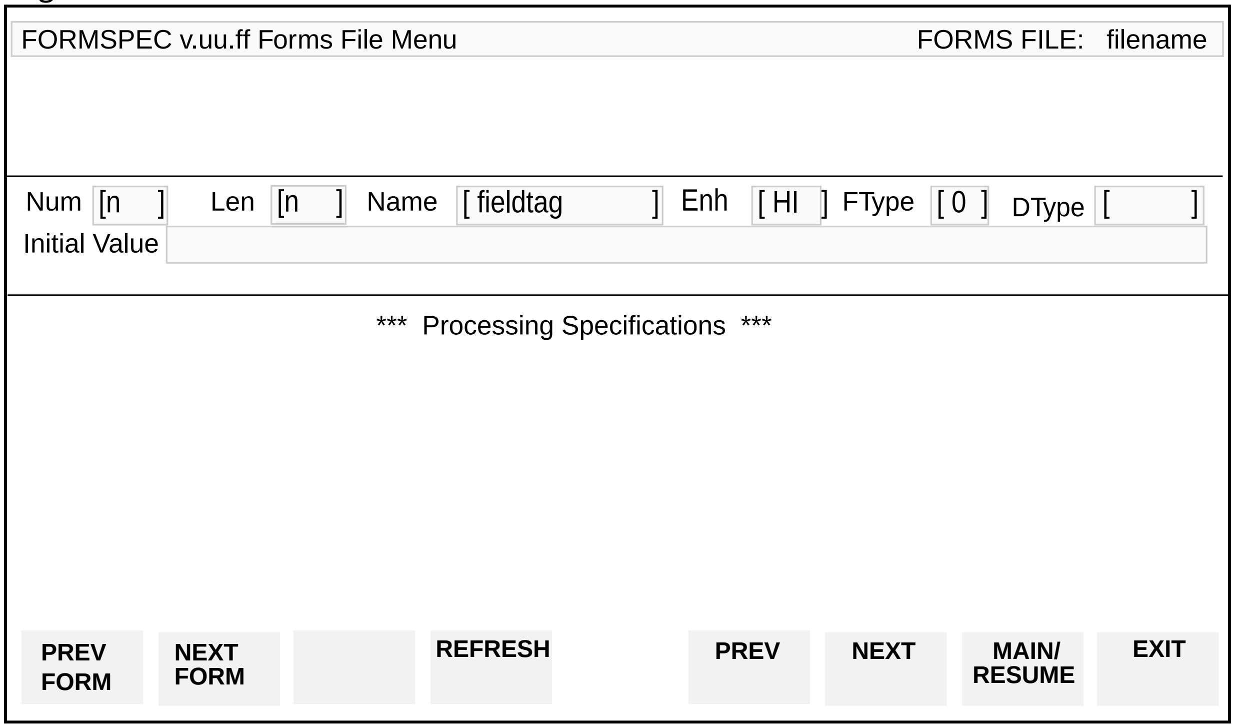
FIELD MENU

In case you want to assign special field characteristics, FORMSPEC displays a Field Menu, illustrated in Figure 3-21., for each field delimited during form design. Each menu displays up to three lines of the form, and marks the particular field to which the menu applies. It then displays the default field attributes of the field. The Field Menus are presented in the order the fields appear on the form layout, from left to right, top to bottom.

You can change the default field attributes by typing new values and pressing **ENTER**. If you do not want to change the defaults, simply press **NEXT** to display the next Field Menu. If you are designing simple forms, in which all data is accepted and no special edits are performed, you can skip all the Field Menus by pressing **NEXT FORM**. If you change the field length or type, the **screen length or type** of the associated ARB (if any) is updated at the same time, but the **ARB length and type** are not changed.

The field attributes, all except length, can also be changed using FORMSPEC in batch mode. See Section 7 for more information. Processing specifications can be specified on the Field Menu. See Section 4 for a description.

Figure 3-21. Field Menu


FORMSPEC v.uu.ff Forms File Menu FORMS FILE: filename

Num [n] Len [n] Name [fieldtag] Enh [HI] FType [0] DType [ ]
Initial Value

*** Processing Specifications ***

Fields

`Num`	(Display only) A number between 1 and 256 assigned by FORMSPEC to the field at form layout. The number is permanently assigned to the field and is not changed even if other field attributes or the position of the field on the form layout changes. However, if the field tag is changed, a new field number is assigned to the field. The field numbers of a form can be

renumbered in screen order with the FORMSPEC batch mode command RENUMBER, as described in Section 7.

Len	(Display only) The length of the field as determined during form layout.
Name	Specify any USASCII name of up to 15 characters long; it must start with an alphabetic character and may be followed by any alphanumeric (A -Z, a-z, 0-9) or the underline (). It cannot be one of the reserved words listed earlier in Figure 3-2.
Enh	Specify any combination of enhancements using up to four characters:

H	Half Bright
I	Inverse video
U	Underline
B	Blinking
S	Security
1-8	Color (refer to Figure 3-7.)
NONE	No enhancement

Default HI

FType — Specify the field type:

D	Display only — cannot be modified by user.
R	Required — cannot be left blank.
O	Optional — may be left blank; edit checks and field phase processing specifications are skipped if the field is blank.
P	Process — may be left blank, but edit checks are performed and field phase processing specifications are executed even if the field is blank.

Default O

DType — Specify the data type:

CHAR	Specifies data to be a string of any characters.
NUM[*n*]	Specifies data to be numeric. The maximum number of decimal places is indicated by the optional digit (*n*), where *n* is a value between 0 and 9. An optional leading sign (+ or -) is allowed; as are commas and decimal points if correctly placed.
DIG	Specifies data to be a positive integer. No sign, no comma, no decimal point allowed.
IMP*n*	Specifies data to be numeric with an assumed decimal point. The assumed number of decimal places is indicated by the required digit (*n*), where *n* is a value between 0 and 9. A sign and commas are allowed. A decimal point is allowed but should not be entered.

`MDY`	Specifies data to be a date in the order month, day, year.
`DMY`	Specifies data to be a date in the order day, month, year.
`YMD`	Specifies data to be a date in the order year, month, day.

When Native Language Support is used, the symbols used to indicate thousands and decimals are language-dependent. The names of months and their abbreviations are accepted for each native language at run-time. For more information on Native Language Support, see Section 8.

Default CHAR

`Initial Value` Specify data to be displayed in the field when the form is initially displayed.

GLOBALS MENU

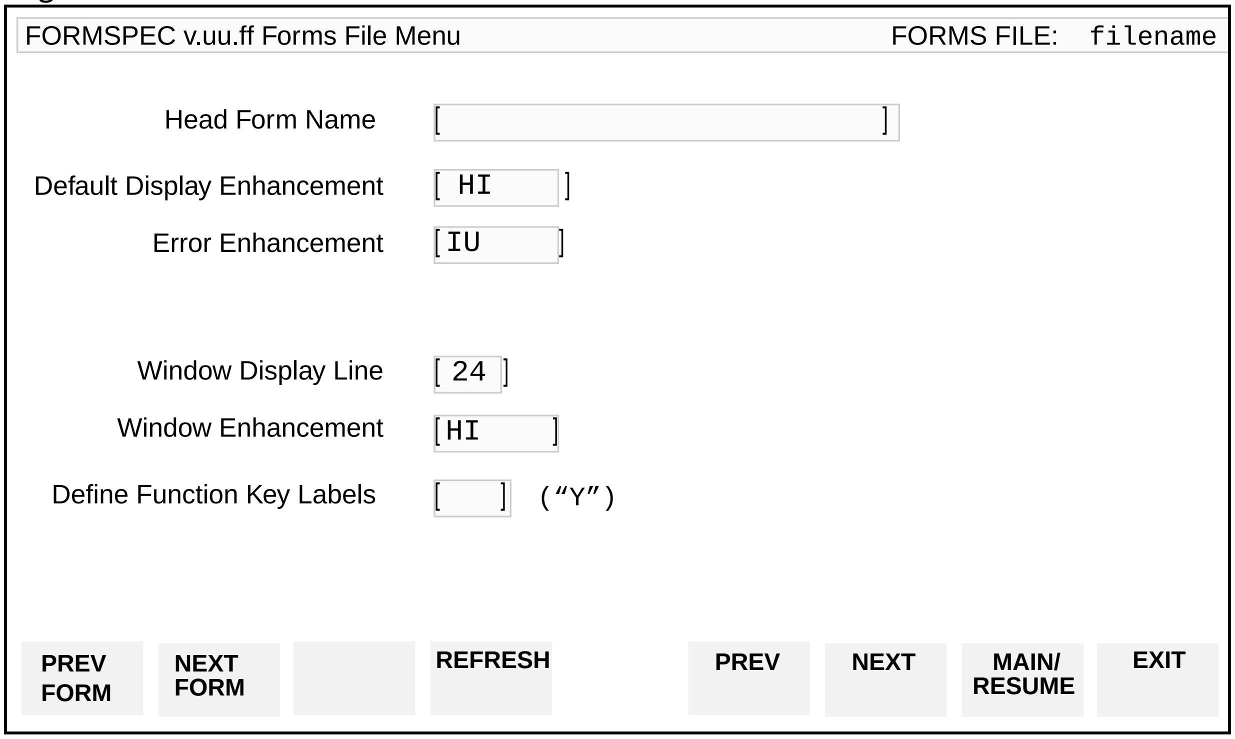
Certain characteristics of the entire forms file are defined as global specifications and specified on the Globals Menu, illustrated in Figure 3-22. The global specifications include the default screen enhancements for fields and errors, the first form to be executed, where the message "window" line appears on the form, and provides for user-defined labels for the function keys. These specifications apply to the entire forms file rather than to individual forms or to fields within the forms.

FORMSPEC supplies default values for these characteristics and, unless you want to change the defaults, you need never be concerned with global specifications. If, however, you want to change these specifications, you enter `G` in the Main Menu selection box, and do not specify a form name or a field name. The Globals Menu is then displayed so you can change the default global specifications. If you indicated the data capture devices on the Terminal/Language Selection Menu, the following message will appear at the bottom of the Globals Menu:

```
Press NEXT to select HP 3075/6 device specifications
```

(The Data Capture Device Specifications Menu is described later in this section).

Figure 3-22. Globals Menu



```
FORMSPEC v.uu.ff Forms File Menu                    FORMS FILE:   filename

              Head Form Name          [                              ]

    Default Display Enhancement       [ HI     ]

              Error Enhancement       [IU      ]


            Window Display Line       [ 24 ]

            Window Enhancement        [HI      ]

     Define Function Key Labels       [    ]   ("Y")
```

When you press **ENTER**, FORMSPEC checks and records the global values (default or specified) and then displays the next menu. The next menu is the Save Field Menu. To get to the first Form Menu, press **NEXT FORM**.

Fields

`Head Form Name`

Enter the name of the first form you want displayed when the forms file is executed. If you leave this field blank, the first form in the forms file will be the first form displayed. Note that adding forms to the forms file may change the first form. When the forms file is compiled, this name must identify an existing form. If you enter the name using lowercase letters, FORMSPEC upshifts the letters to uppercase and then looks for a matching form name.

Default First form in USASCII collating sequence in the forms file. (`$HEAD`)

`Default Display Enhancement`

When defining a form, you can specify individual display enhancements for any field on the Field Menu. If you do not specify such enhancements on the Field Menu, FORMSPEC assigns default field enhancements. If you do not want to use these default display enhancements, you can specify your own default enhancements here. Enter one or more of the display enhancement codes (`B, H, I, S, U`, and `1 - 8`) in any combination, or you can enter `NONE` if you do not want the fields to be enhanced at all. The enhancement codes can be entered in any combination, in any order. For example, for blinking, half bright, underlined, you can specify `BHU, HUB`, `UBH`, and so forth. If you want to remove all display enhancements, enter `NONE`. The data capture devices do not support display enhancements.

Default Half bright, Inverse video (`HI`).

`Error Enhancement`

When a user entering data on a form makes an error, the field with the error is highlighted on the form by special display enhancements. You can change the default error enhancement by entering one or more of the display enhancement codes (`B, H, I, S, U`, and `1 - 8`) in any combination, or you can enter `NONE` if you do not want fields with errors to be enhanced at all. For example, if you want fields with errors to be displayed in half bright, underlined, and to blink, enter `BHU`.

Default Inverse video, full bright, Underline (`IU`).

`Window Display Line`

You can specify the line of the terminal screen to be reserved for error and status messages. Screen lines are numbered from 1 (top) through 24 (bottom). You may enter 0 (zero) if you do not want a window. Note that in this case no error or status messages are displayed on the form during execution. Any fields in which errors are detected will still be enhanced when there is no window line, unless you specify `NONE` for error enhancements. If function key labeling on HP 264X terminals is in effect and a window line of 23 or 24 is chosen, line 22 is actually used.

Default Bottom line (24)

`Window Enhancement`

The window line is normally enhanced with inverse video, half bright. If you want a different enhancement for this line, you must specify it in this box. You may enter any of the standard enhancement codes (`I, H, U, B, S,` and `1 - 8`) in any combination, or you can specify no enhancements by entering `NONE`.

Default Inverse video, Half bright (`HI`).

`Define Function Key Labels`

To define function key labels, enter `Y` to obtain the Global Function Key Labels Menu. If you wish to retain the default global labels, leave this box blank. When function key labels have been defined, FORMSPEC will place an `X` in the Define Function Key Labels field. Once function key labels have been defined, the default labels can be reset by entering an `N` in this field. If you specify a `Y` in this field, the Global Function Key Labels Menu is displayed, as shown in Figure 3-23.The data capture devices do not support function key labeling.

`Define Data Conversions`

To use the ARB feature, you must specify default conversions for screen data to application data and vice versa. Type `Y` in this field to reach the Data Type Conversions Menu, shown in Figure 3-27. The default data type conversions must be set up before you can create an ARB for a form.

GLOBAL FUNCTION KEY LABELS MENU

This menu is used to specify new function key labels which will appear when an application is executed. Labels you specify replace original labels provided by VPLUS.

Each label consists of two lines of eight characters each. To specify a label, enter the first line in the first field and the second line of the label in the second field in the menu. The color pairs fields allow you to specify a number, (`1 - 8`) to indicate a color for each function key; refer to Figure 3-23. for the correct color to number mapping.

For example, to change the label for Key 1 to read:

ORDER
NUMBER

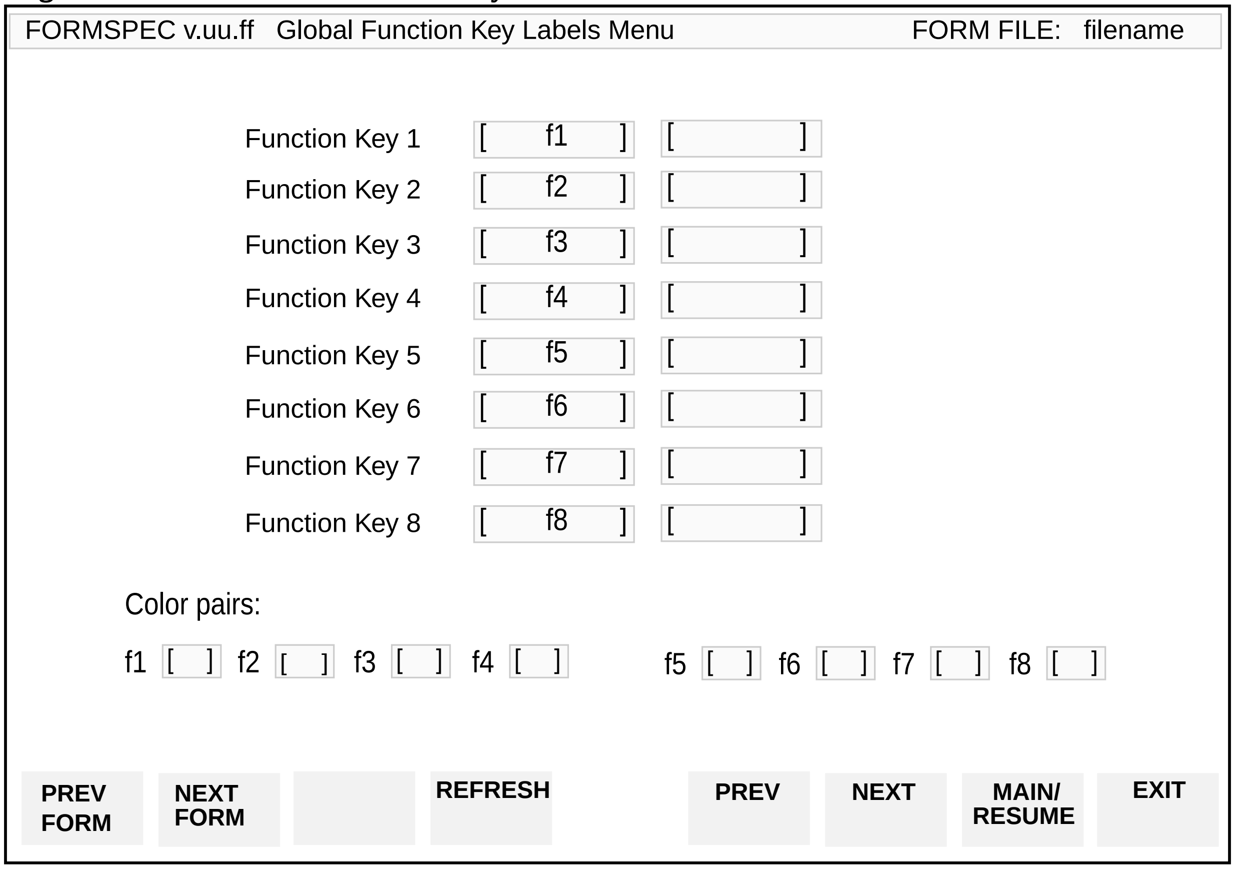
Change the field for Key 1 as shown:

Function Key 1 [[

If the labels are to be Global, they will be displayed for all forms that do not have Local labels defined.

In the case of a frozen form with another form appended to it, the labels displayed will be those for the appended form. If the appended form has no Local labels, the file's Global labels will be used.

Figure 3-23. Global Function Key Labels Menu



FORMSPEC v.uu.ff Global Function Key Labels Menu FORM FILE: filename

Function Key 1	[f1]	[ ]
Function Key 2	[f2]	[ ]
Function Key 3	[f3]	[ ]
Function Key 4	[f4]	[ ]
Function Key 5	[f5]	[ ]
Function Key 6	[f6]	[ ]
Function Key 7	[f7]	[ ]
Function Key 8	[f8]	[ ]

Color pairs:

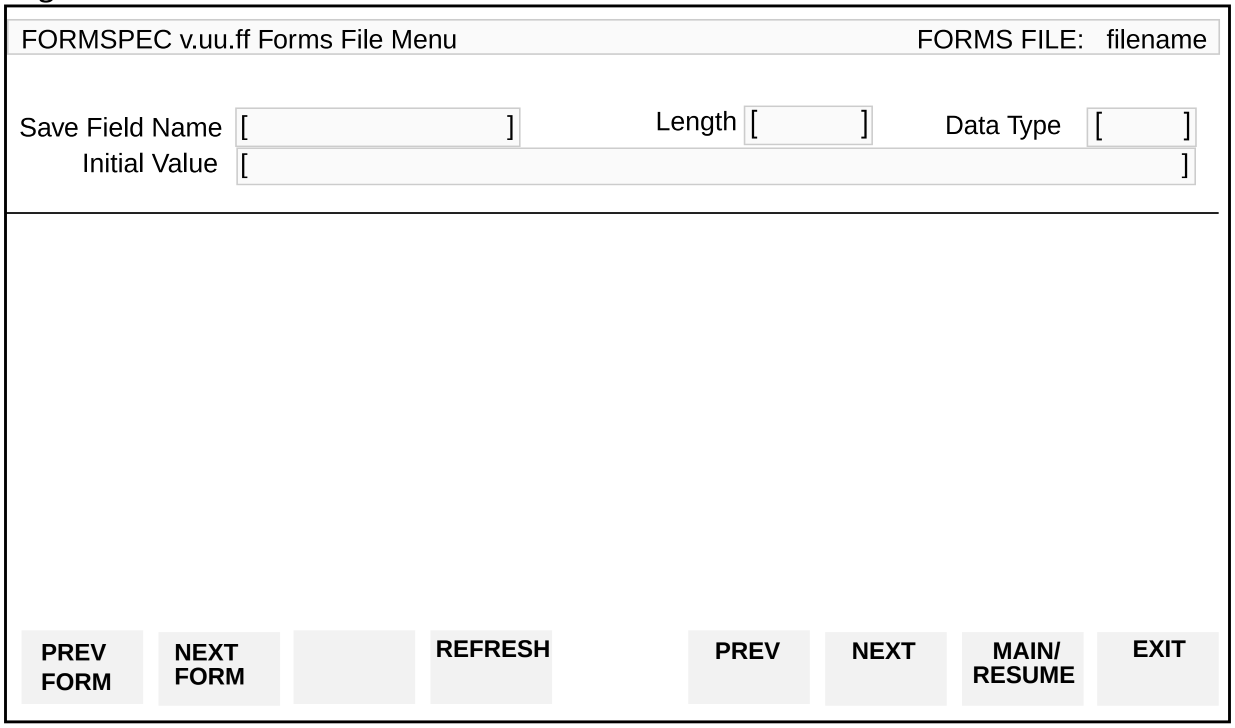
f1 [ ] f2 [ ] f3 [ ] f4 [ ] f5 [ ] f6 [ ] f7 [ ] f8 [ ]

SAVE FIELD MENU

A save field is a global field whose value can be used anywhere in the forms file where field references are allowed. Save fields are not part of a particular form like other fields, but are "global" to all forms. Like other fields, they always contain data in external, character form. The save fields can be used by most processing statements, but are particularly useful for passing data between fields on different forms. Refer to the SET command in Section 4. A maximum of 20 save fields can be defined for any forms file.

Save fields are specified on the Save Field Menu, which is displayed when requested on the Main Menu. Any defined save fields are stored at the beginning of the forms file just after the global specifications. (Refer to "Forms File" earlier in this section for the order of specifications in the forms file.) The Save Field Menu is illustrated in Figure 3-24.

Figure 3-24. Save Field Menu



FORMSPEC v.uu.ff Forms File Menu FORMS FILE: filename

Save Field Name [ ] Length [ ] Data Type [ ]
Initial Value [ ]

A save field name appears exactly the same as a field name. To help distinguish the global save fields from local fields, you may want to establish a naming convention. For example, if all save field names (and no local field names) start with the letter SF, then any field name starting with SF is immediately recognizable as a save field.

Up to 20 save fields can be defined in one forms file. The Save Field Menu is usually requested through the Main Menu. To add a save field, press MAIN/RESUME to request the Main Menu and then type S in the selection box and press **ENTER**. A blank Save Field Menu is then displayed. You can also display a blank Save Field Menu by pressing PREV from the first Form Menu.

To delete a save field, you also request the Main Menu and then enter D in the selection box and the name of the save field to be deleted in the field name box.

Fields

`Save Field Name`	A standard field identifier composed of up to 15 USASCII characters, beginning with an upper or lowercase letter. The remaining characters can be upper or lowercase letters, digits, or underlines. For example, `SF3`, `D2 15`, or `ALPHA`. Note that if you use a lowercase letter in a name, FORMSPEC shifts it to an uppercase letter. (The name must not be one of the reserved words listed earlier in Table 3-2.
`Length`	Like other fields, save fields must be assigned a length, representing the maximum number of characters allowed in the field. Note that you must supply a field length, unlike field specifications where field length is determined from the screen design. Since save fields may be used as accumulators, it is important that the save field length be long enough to avoid rounding and/or truncation when summed values are moved to the field.
`Data Type`	The data types allowed are the same as for any field. The general types are character (CHAR), numeric (NUM[n], IMPn, or DIG), or date (MDY,YMD, or DMY). (Refer to "Ease of Forms Design" for a complete discussion of these data types.)

`Initial Value`	An initial value may be assigned to any save field. This is an optional specification; if omitted, all fields are set to blanks (`$EMPTY`). If specified, the initial value must be the same type as the specified data type. Values are entered exactly as if they were entered by a terminal user; that is, characters are not delimited by quotes, and dates are not delimited by exclamation points. For example:
	`12.5` — `type is NUM1`
	`3790` — `type is DIG`
	`FEB 3, 86` — `type is MDY`
	`John` — `type is CHAR`
	When the forms file is opened, initial values (default or specified) are assigned to the save fields.

USING FORMSPEC TO CREATE AN ARB

Once you have created a form and saved it in a forms file, you can create an associated application-ready buffer (ARB) for that form. There are two steps to the process; first, you must set up a data type conversion (DTC) record, then you can generate the ARB. The DTC record need only be set up once for the entire forms file.

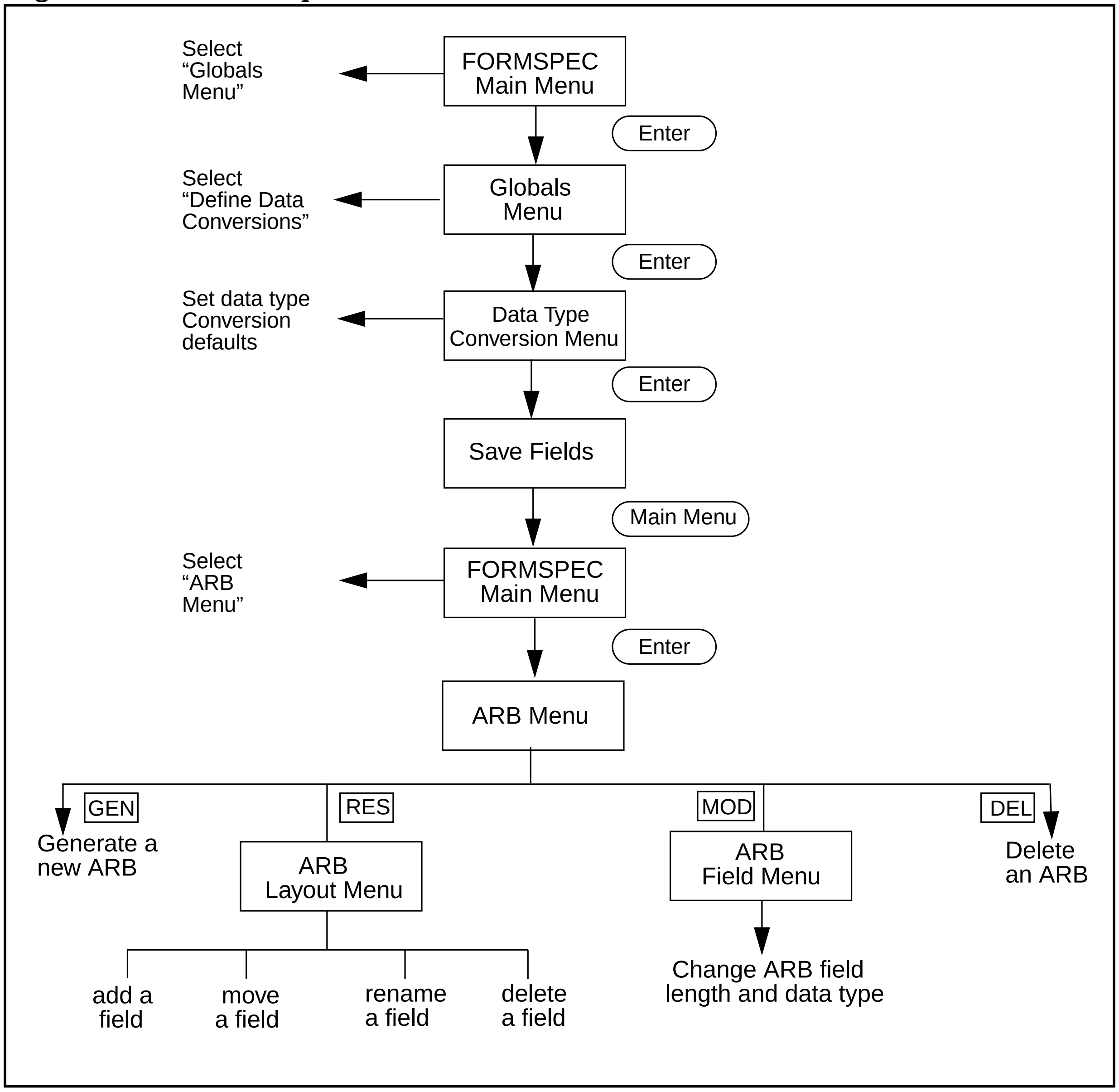
You may want to transform data between the screen and application for several reasons. First, the data the application will store may differ from what appears on the screen: menu-selection and next-screen fields, for example, would normally be excluded from the ARB. Conversely, you may want to store data from a source other than the screen, such as key fields for an IMAGE dataset, along with the screen data. This can be done by using "filler" fields, fields that exist on the ARB but not on the corresponding form (see RES below).

Second, the order in which the application stores data may differ from the order in which it is entered on the screen. An arrangement of fields that makes logical sense to the user may not be suitable for a database, for example. Fields can be rearranged on the screen without affecting their order on the ARB, and vice versa.

Third, data type and length may differ from screen to ARB, and the designer uses the ARB screens, in conjunction with the Data Type Conversions Menu, to specify conversions (see discussion under "ARB Data Types").

Figure 3-25. shows the sequence of menus used to create, modify and delete an ARB and the fields on it.

Figure 3-25. Menu Sequence for ARB Feature



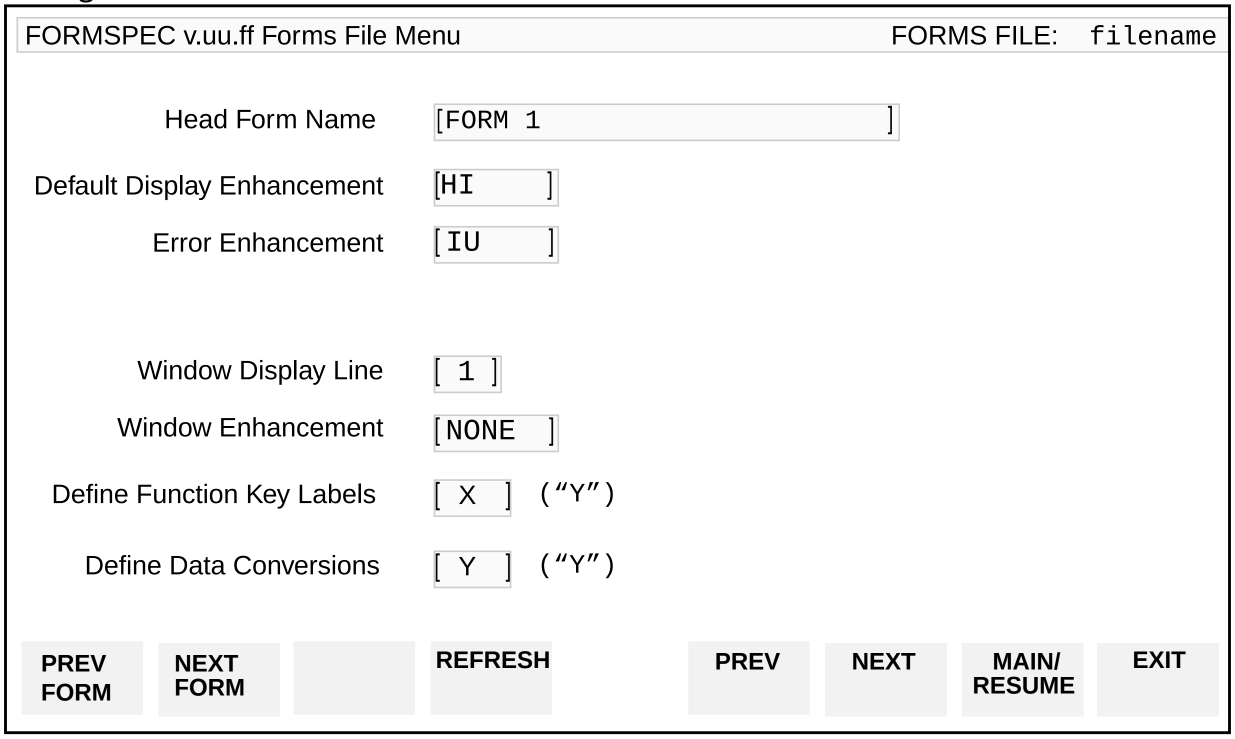
Setting Up the Data Type Conversion Record

The data type conversion record allows the forms designer to define the default values for screen and ARB data types. You must define these values, even if you only press **ENTER** to accept FORMSPEC's preset defaults, shown in Figure 3-27. You will probably want to replace some, if not all, of these defaults with your own.

Set up the data type conversion record by following these steps:

1. Make sure you are at the FORMSPEC Main Menu and select **G** for "Go to Globals Menu". when you press **ENTER**, the Globals Menu is displayed.

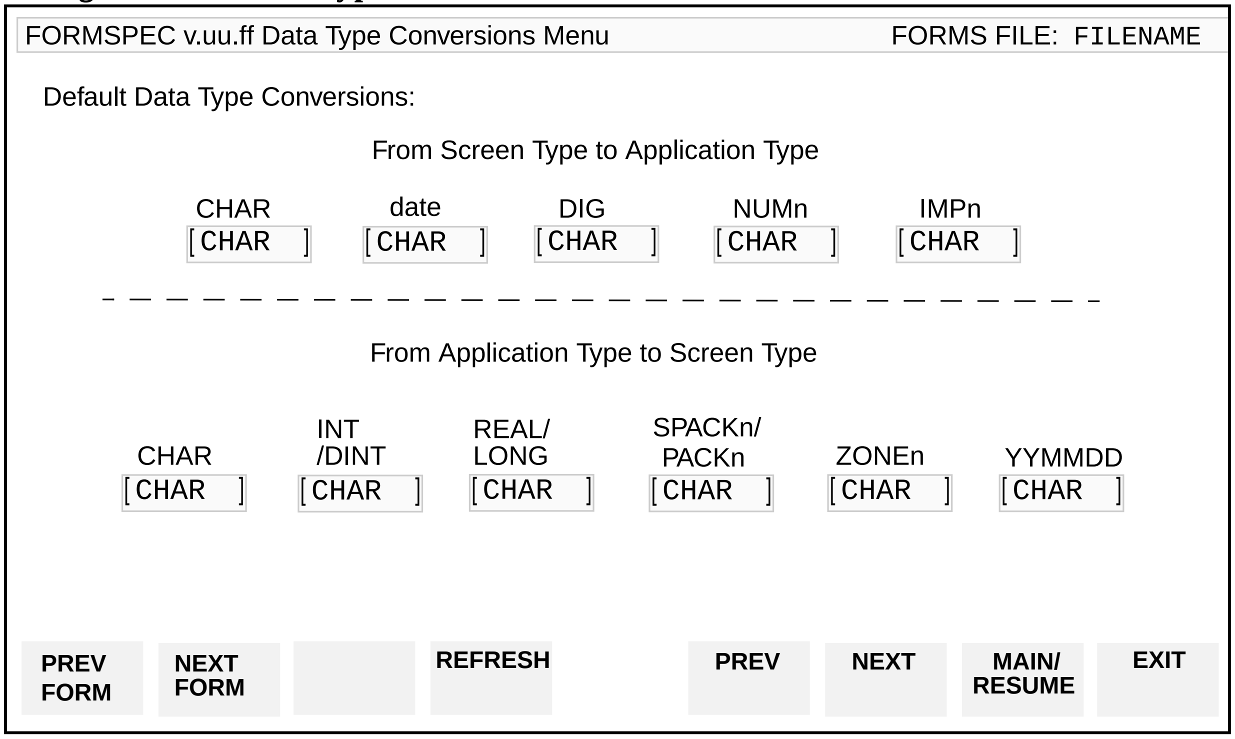
Figure 3-26. Globals Menu



```
FORMSPEC v.uu.ff Forms File Menu                        FORMS FILE:   filename

              Head Form Name          [FORM 1                         ]

  Default Display Enhancement         [HI     ]

            Error Enhancement         [IU     ]


          Window Display Line         [ 1 ]

          Window Enhancement          [NONE   ]

   Define Function Key Labels         [ X  ]  ("Y")

     Define Data Conversions          [ Y  ]  ("Y")

PREV    NEXT            REFRESH          PREV    NEXT    MAIN/    EXIT
FORM    FORM                                             RESUME
```

2. Enter **Y** for "Define Data Conversions" in the last box. The Data Type Conversions menu is displayed. This screen allows you to specify default conversions to and from every available screen and application data type.

Figure 3-27. Data Type Conversion Menu



```
FORMSPEC v.uu.ff Data Type Conversions Menu             FORMS FILE:  FILENAME

  Default Data Type Conversions:

                      From Screen Type to Application Type

          CHAR         date        DIG        NUMn        IMPn
        [CHAR  ]    [CHAR  ]    [CHAR  ]    [CHAR  ]    [CHAR  ]
    - — — — — — — — — — — — — — — — — — — — — — — — — — — — -

                      From Application Type to Screen Type

                  INT        REAL/      SPACKn/
       CHAR       /DINT      LONG        PACKn      ZONEn      YYMMDD
     [CHAR  ]   [CHAR  ]   [CHAR  ]    [CHAR  ]   [CHAR  ]   [CHAR  ]


PREV    NEXT            REFRESH          PREV    NEXT    MAIN/    EXIT
FORM    FORM                                             RESUME
```

The *n* in the numeric types stands for a number of decimal places. When specifying a conversion to one of these types, you must replace *n* with a digit or *a*. When you set the number of decimals to *a*, you are instructing FORMSPEC to determine the decimals algorithmically on the basis of the source type. Note that you can only select the *a* option on this menu, and not on the field menus. For recommendations on data type conversions, see "Data Types" earlier in this section.

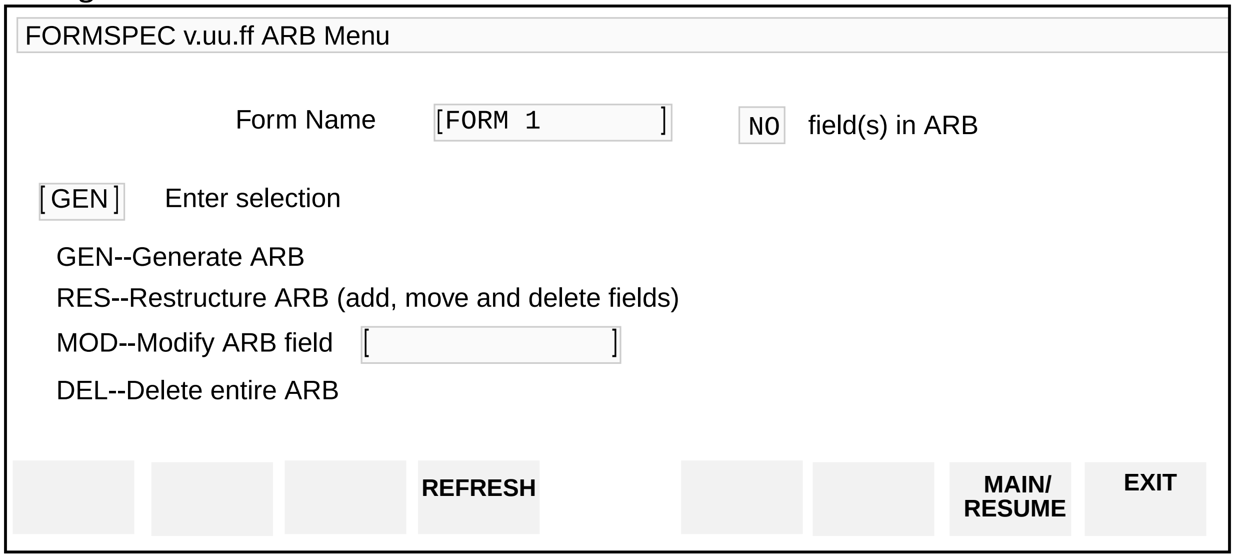
3. When you have set your defaults, press **ENTER** and then press MAIN/RESUME at the SAVE FIELD Menu to return to the Main Menu.

Generating the ARB

Follow these steps to generate an ARB for a form.

1. At the Main Menu, enter **B** for "Go to ARB Menu" in the Main Menu Selection box.
2. In the "Go to ARB Menu" field, enter the name of the form for which you wish to generate an ARB and press **ENTER**. The ARB Menu is displayed.

Figure 3-28. ARB Menu



FORMSPEC v.uu.ff ARB Menu

Form Name [FORM 1] NO field(s) in ARB

[GEN] Enter selection

GEN--Generate ARB

RES--Restructure ARB (add, move and delete fields)

MOD--Modify ARB field [ ]

DEL--Delete entire ARB

The form name is displayed in the first field. This is a display-only field: you cannot select a new form from this menu. If you want to create an ARB for a different form, you must go back to the Main Menu.

The next field shows the number of fields in the ARB: in this case there are none, since there is no ARB yet.

Types of Field

An ARB may contain two kinds of field. They are live fields and filler fields.

- Live fields exist on both the screen and the ARB.
- Filler fields exist on the ARB only; for example, alignment and accumulator fields.

The designer can select one of four options on the ARB Menu.

Options	
`GEN (Generate)`	Create an ARB from scratch, containing a field for each field on the form. The sequence of the fields accords with their sequence on the form, and can be altered using the Restructure ARB Menu Figure 3-29. The type and length of the fields are determined by comparing the screen attributes with the data conversion record. They can be changed using the ARB Field Menu. This option is only valid if no ARB exists for the form.
`RES (Restructure)`	Display the current ARB, showing the sequence of fields. Allows the designer to add, delete, move and rename fields using the Restructure ARB Menu.
`MOD (Modify)`	Modify the length and data type of the designated field in the ARB, using the ARB Field Menu.
`DEL (Delete)`	Delete the entire ARB. If you want to make a lot of changes to the fields on a form, it is usually better to delete the existing ARB and generate a new one once the form has been altered. If you copy or delete a form, the corresponding ARB will be copied or deleted as well.

Table 3-11. shows how an alteration made to the form will affect the ARB, if at all.

Table 3-11. Form/ARB Relationships

Form	**ARB**
Adding a form...	Does not add an associated ARB.
Copy a form...	Copies the associated ARB.
Renaming a form...	Renames the associated ARB.
Deleting a form...	Deletes the associated ARB.
Changing a form: - adding a field...	Does not add it to the ARB once the ARB has been created, unless you create a new ARB.
- changing a field...	Changes the screen length and screen data type on the ARB, but does not affect the ARB field length and data type.
- deleting a field...	Changes the corresponding ARB field to a "filler" field.
No effect on form...	Add, change delete ARB or the fields in it.
Form can exist without an ARB.	ARB cannot exist without a form.

The Effect of Renaming a Field on a Form There are three possibilities, which are described below.

- The old name is in the ARB but the new name is not: The ARB field is renamed to the new name, retaining its "live" field type, and the screen length and screen type are altered if necessary.
- The old name is not in the ARB, but the new name is, as a "filler" field: The ARB field changes from "filler" to "live" field type, and the screen length and screen data type are updated if necessary.
- Both old and new names exist in the ARB: The *old* name changes from "live" to "filler" type, and the screen length and screen data type change to zero and blanks respectively. The *new* name changes from "filler" to "live" field type, and the screen length and screen data type are updated.

NOTE FORMSPEC allows the designer to delete a field from a form after a corresponding field has been added to the ARB. The field need not be deleted from the ARB as well; at compile-time, the ARB field is converted to a "filler" field, with screen length set to zero and screen type blank. The designer can; however, choose to delete or rename the ARB field.

It is possible, though usually not advisable, to create an ARB without using the GENerate feature. To do this, type RES and use the Restructure ARB Menu to add fields one at a time.

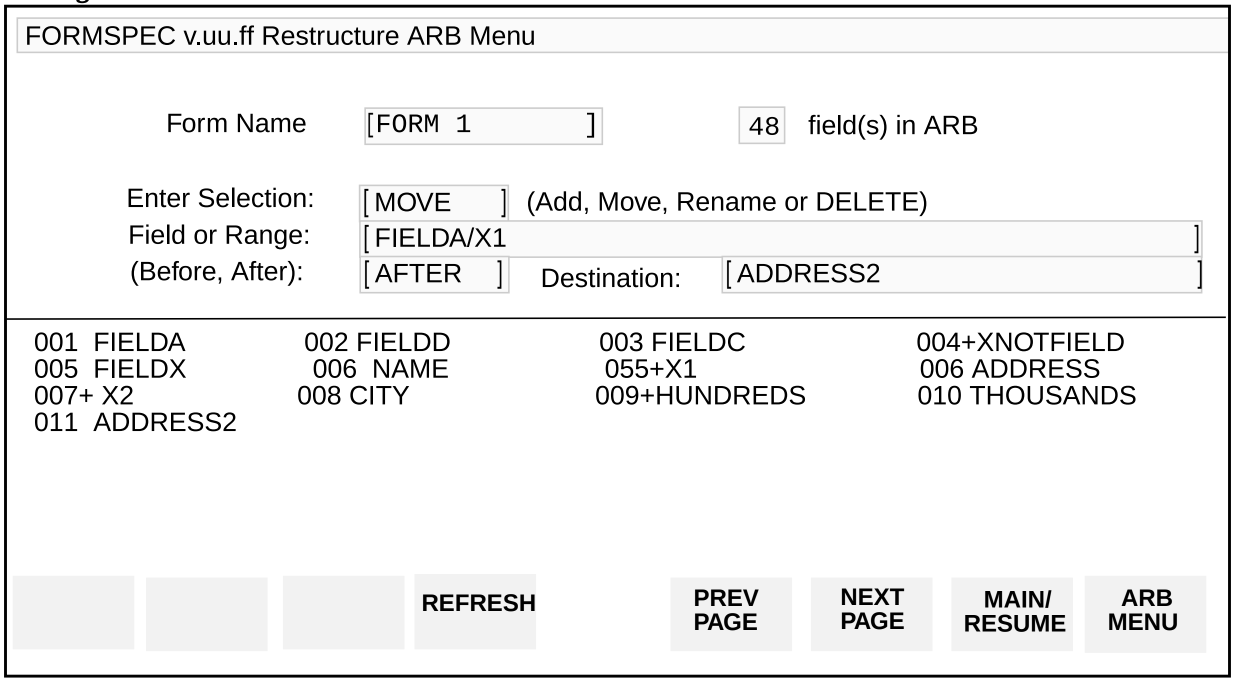
If your default conversion for any field(s) could cause run-time errors, you will get a message to that effect (see Appendix B, *VPLUS ERROR MESSAGES*). Go to the ARB Field Menu to check that the conversion does make sense in this case, or to change the target datatype if it doesn't.

3. Enter GEN in the Selection box and press **ENTER** to generate an ARB for this form. The "No field(s) in ARB" changes to show a number equivalent to the number of fields on the form.
4. You can now change the position of a field in the ARB, add or delete fields, or modify a field by changing its length and/or data type.
5. To move, rename, add or delete fields, type RES in the Selection box and press **ENTER**. The Restructure ARB menu is displayed.

   This menu displays all the fields in the ARB by name and in sequence. The form name is displayed in the first field (display only). The lower part of the screen shows the ARB fields. Use PREV PAGE and NEXTPAGE to view them all if necessary.

NOTE ARB changes do not propagate to form family members. Changes impact the current form only.

Figure 3-29. Restructure ARB Menu



```
FORMSPEC v.uu.ff Restructure ARB Menu

        Form Name      [FORM 1       ]          48  field(s) in ARB

   Enter Selection:   [MOVE    ]  (Add, Move, Rename or DELETE)
   Field or Range:    [FIELDA/X1                                        ]
   (Before, After):   [AFTER   ]   Destination:  [ADDRESS2              ]

 001  FIELDA        002 FIELDD        003 FIELDC        004+XNOTFIELD
 005  FIELDX         006  NAME         055+X1           006 ADDRESS
 007+ X2            008 CITY          009+HUNDREDS      010 THOUSANDS
 011  ADDRESS2
```

6. Use the fields at the top of the screen to change the order by moving fields or ranges. Press **ENTER** to execute all changes.

Options `Command`

Must state `MOVE, ADD, RENAME`, or `DELETE`. `DELETE` must be spelled out; the other commands may be abbreviated.

`Field or Range`

Name(s) or numbers(s) of the field(s). A slash (/) indicates a range: valid for `MOVE` or `DELETE` only. Any field may be deleted.

You can `ADD` only a single field at a time; it may not already exist on the ARB and it must conform to FORMSPEC naming rules. You may add a field that does not exist on the associated form. This "filler" field will have an ARB default length of 1 byte, and ARB type of CHAR; screen length is zero and screen type is blank. Fillers are marked with a "+" on the Restructure ARB Menu.

To `RENAME` a field, enter the name of an existing field, and enter the new name of the field in `DESTINATION`. "Live" fields exist on both form and ARB; "filler" fields exist on the ARB only. The following defaults apply to renamed ARB fields:

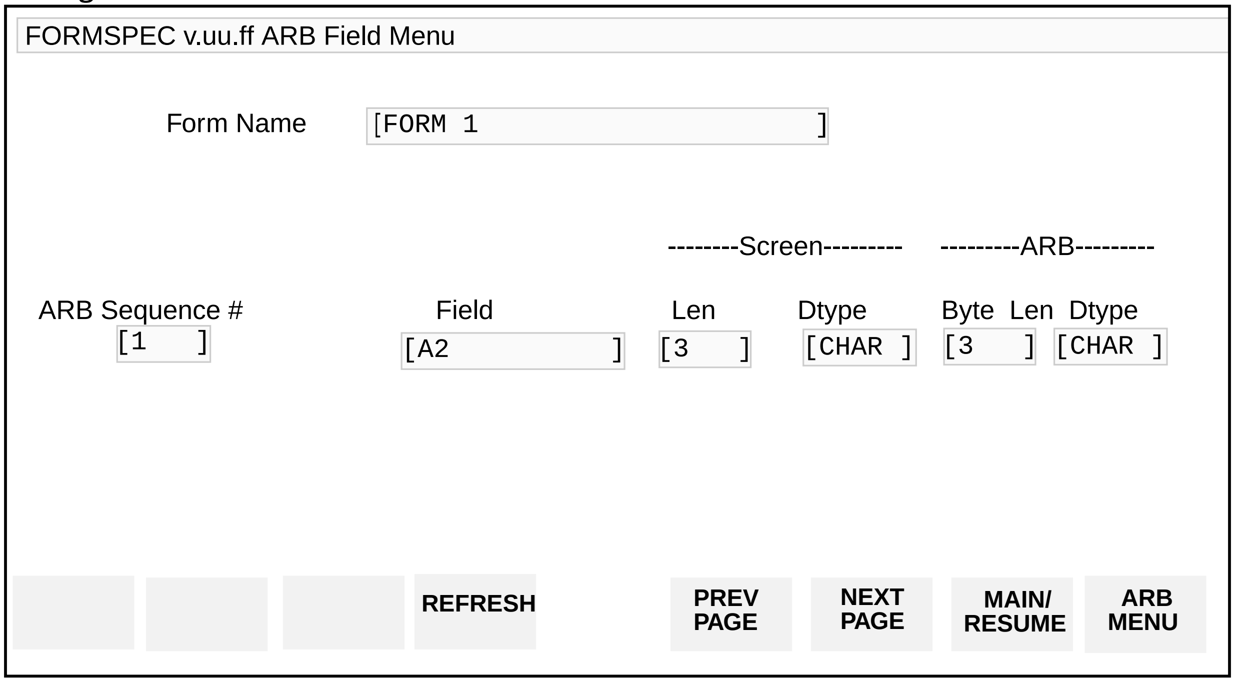
- **Live name changed to live name** (new ARB field has corresponding field on the form): new ARB field length and type derived algorithmically from the length and type of the corresponding field on the form.
- **Filler name changed to live name** (new ARB field has corresponding field on the form): new ARB field length and type derived

algorithmically from the length and type of the corresponding field on the form.

- **Live name changed to filler name** (new ARB field has no corresponding field on the form): new field retains length and type of old ARB, and screen length becomes zero and type blank.
- **Filler name changed to filler name** (new ARB field has no corresponding field on the form): new field retains length and type of old ARB, and screen length remains zero and type blank.

7. When you have finished modifying the field order on the ARB, press **ARB MENU** to return to the ARB Menu, or **MAIN/RESUME** to go back to the Main Menu.
8. To change the length or data type of an ARB field, type `MOD` on the ARB Menu. The ARB Field menu is displayed.

Figure 3-30. ARB Field Menu



FORMSPEC v.uu.ff ARB Field Menu

Form Name [FORM 1]

--------Screen--------- ---------ARB---------

ARB Sequence #	Field	Len	Dtype	Byte	Len	Dtype
[1]	[A2]	[3]	[CHAR]	[3]		[CHAR]

This screen allows you to scroll through all the fields in the ARB and change their length and data type if required.

NOTE You can change only the ARB field type and length. To change the characteristics of the field on the form, you must use the FORMSPEC Field Menu (see figure 3-22). If you use this menu to change the length or type of a field on the form, the corresponding screen length and screen type of the field in the ARB will be changed accordingly, but the ARB length and ARB type will remain the same.

Using and Defining ARBs The programmer is responsible for:

- Using the correct data types for each programming language; for example, REAL is invalid for COBOL applications.
- Aligning and/or padding the ARB/source code. If an odd number of bytes is followed by an integer, some languages, including Pascal and FORTRAN, automatically pad the record definition, forcing the integer to be word-aligned. No such padding occurs in COBOL unless the SYNCHRONIZED clause is used (see the *COBOL II/3000 Reference Manual).*

If you use a language whose compiler pads to ensure word alignment for integers, you must pad your ARB correspondingly. For example, suppose you are coding in Pascal and you declare a record that looks like this:

```
ODD_BYTE_EXAMPLE = RECORD
  THREE_BYTES    : PACKED ARRAY [1..3] OF CHAR;
   GOTCHA         : INTEGER
     .
     .
   END;
```

The Pascal compiler will insert an additional byte after THREE_BYTES to align the integer on a word boundary. You must do the same with the ARB record; use the ARB LAYOUT screen to add a filler field after the three-byte field so that the ARB looks like this:

Field Name	*ARB Type*	*ARB Length*
DEPT	CHAR	3
FILLER1	CHAR	1
TOTAL PURCHASES	DINT	4

- Ensuring the application specifications match the ARB specifications, for example, if the ARB type is PACK, the COBOL specification should be COMP-3, not COMP.
- Avoiding run-time errors, which may occur when:
  - — Converting a CHAR or date source to a numeric destination
  - — Converting a numeric or CHAR source to a date destination
  - — Invalid length specifications are encountered
  - — There are alignment problems; FORMSPEC does not detect these

    There is also the possibility that data will be truncated; for example, if DIG -> INT, but *screenlen* is greater than 5.

The ARB Trace Facility The ARB trace facility may be enabled by setting the JCW **VPLUSARBTRACE** to 1. This will print trace messages to the stdlist. You may direct VPLUS screens to a different device by using a FILE command to set the device of the filename used by VOPENTERM to a device other than the stdlist device.

This is an example of a trace from a form that has seven fields.

```
:SETJCW VPLUSARBTRACE,1
   :FILE VTERM;DEV=99
   :RUN ARBPROG

Field 1    : Buffer offset from 0 = 1
                            Length = 2
                              Type = INT
                             Value = 1234

Field 2    : Buffer offset from 0 = 3
                            Length = 2
                              Type = INT
                             Value = 1234

Field 3    : Buffer offset from 0 = 6
                            Length = 2
                              Type = INT
                             Value = 1234

Field 4    : Buffer offset from 0 = 9
                            Length = 4
                              Type = DINT
                             Value = 12345678

Field 5    : Buffer offset from 0 = 14
                            Length = 4
                              Type = DINT
                             Value = -1234567

Field 6    : Buffer offset from 0 = 19
                            Length = 4
                              Type = DINT
                             Value = 12345678

Field 7    : Buffer offset from 0 = 23
                            Length = 8
                              Type = CHAR
                             Value = ABCDEFGH

END OF PROGRAM
:
```

The trace provides the following information. It shows:

- The location of each field in the ARB (Buffer offset);
- The length of each field in the ARB (Length);
- The type of transformation that occurs (Type);
- The value that is transformed in the ARB field (Value).

NOTE This ARB is tied to seven form fields, but there are gaps in the buffer before field 1, and between each field from 2 to 6. These gaps are one-byte filler fields.

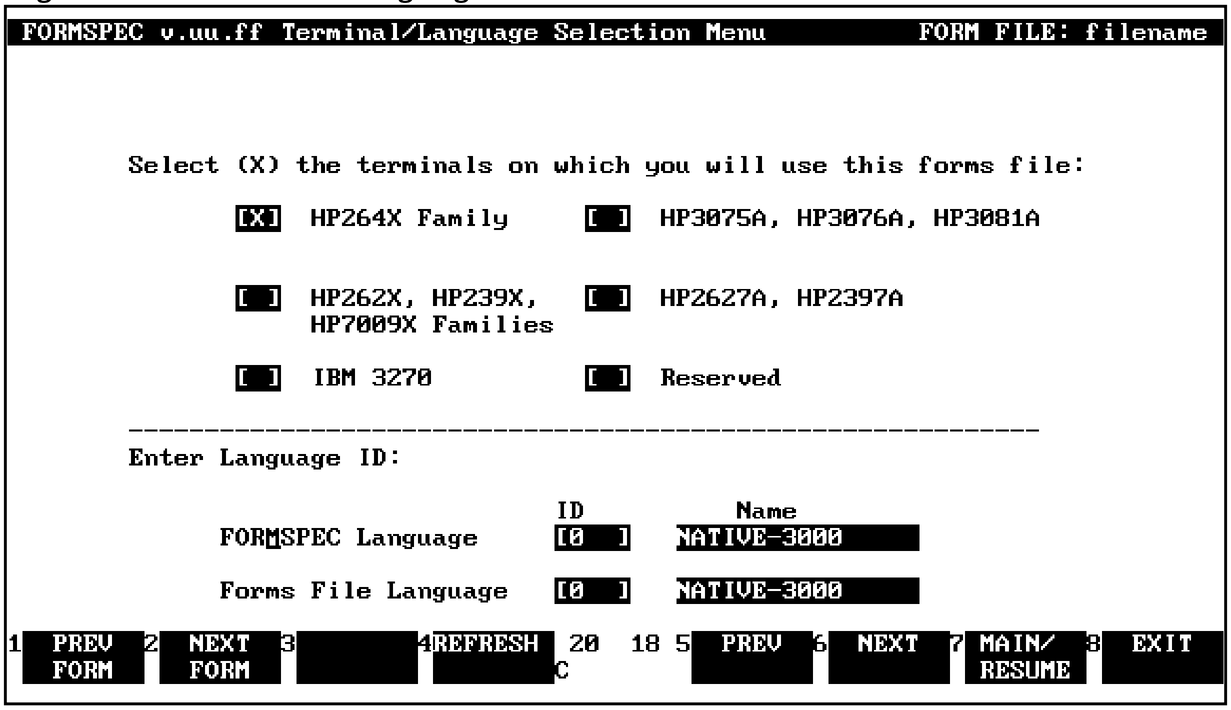
TERMINAL/LANGUAGE SELECTION MENU

The Terminal/Language Selection Menu allows you to specify which terminals will be used with the forms file. You may select one family or any combination of families, including all four families (HP 264X, HP 262X, HP 239X (includes HP 150), and HP 307X). The HP 264X family is the terminal default value. When the selection is defaulted, the forms file will run on the HP 264X family, the HP 262X/HP 239X family and the HP 150, but the following features of selected terminals will not be supported: color, local edits, function key labeling on HP 264X terminals, and security display enhancement. Consult Appendix G to see which HP terminals support which features. The family of terminals need not be specified unless the forms file will be making use of these terminal features or will be used with the data capture devices. Refer to the field descriptions below for more information.

The Terminal/Language Selection Menu also allows you to specify the native language with which the forms file will be used. See "Setting the Language ID Numbers", Section 8, for more information.

The options available on the Terminal/Language Selection Menu are also available using FORMSPEC in batch mode. (See Section 7, for more information.)

Figure 3-31. Terminal/Language Selection Menu



```
 FORMSPEC v.uu.ff  Terminal/Language Selection Menu          FORM FILE: filename



        Select (X) the terminals on which you will use this forms file:

               [X]  HP264X Family       [ ]  HP3075A, HP3076A, HP3081A


               [ ]  HP262X, HP239X,     [ ]  HP2627A, HP2397A
                    HP7009X Families

               [ ]  IBM 3270            [ ]  Reserved

        ------------------------------------------------------------
        Enter Language ID:

                                           ID          Name
              FORMSPEC Language           [0   ]     NATIVE-3000

              Forms File Language         [0   ]     NATIVE-3000

1 PREV   2 NEXT   3        4REFRESH  20   18 5  PREV   6  NEXT   7 MAIN/   8  EXIT
  FORM     FORM                      C                              RESUME
```

Fields

`Select (X) the terminals`

The choices are:

`HP264X Family` Specify an X for HP 264X terminals; specify a Y for HP 264X terminals with function key labeling.

`HP262X, HP239X Families` Specify an `X` for HP 262X and HP 239X terminals; includes HP 150 terminals.

`HP3075A, HP3076A, HP3081A` Specify an `X` in the HP 3075A, HP 3076A, HP 3081A box in order for the forms file to run on the data capture devices; specify an `8` to run on the data capture devices and take full advantage of the HP 3081A.

`HP2627A, HP2397A` Specify an `X` for terminals with the color feature; you must also specify an `X` in the `HP262X, HP239X Families` field.

If you want the forms file to run on the HP 264X and HP 262X/HP 239X families, you must specify *both* on the Terminal/Language Selection Menu.

`Enter Language ID:`

Specify the following:

`FORMSPEC Language` Only `0` for `NATIVE-3000` is supported.

`Forms File Language` Specify one of the following:

`0` for NATIVE-3000

the native language ID number for a language-dependent forms file or

`-1` for an international forms file. Refer to Section 8 for more information.

Default `0` for `NATIVE-3000`

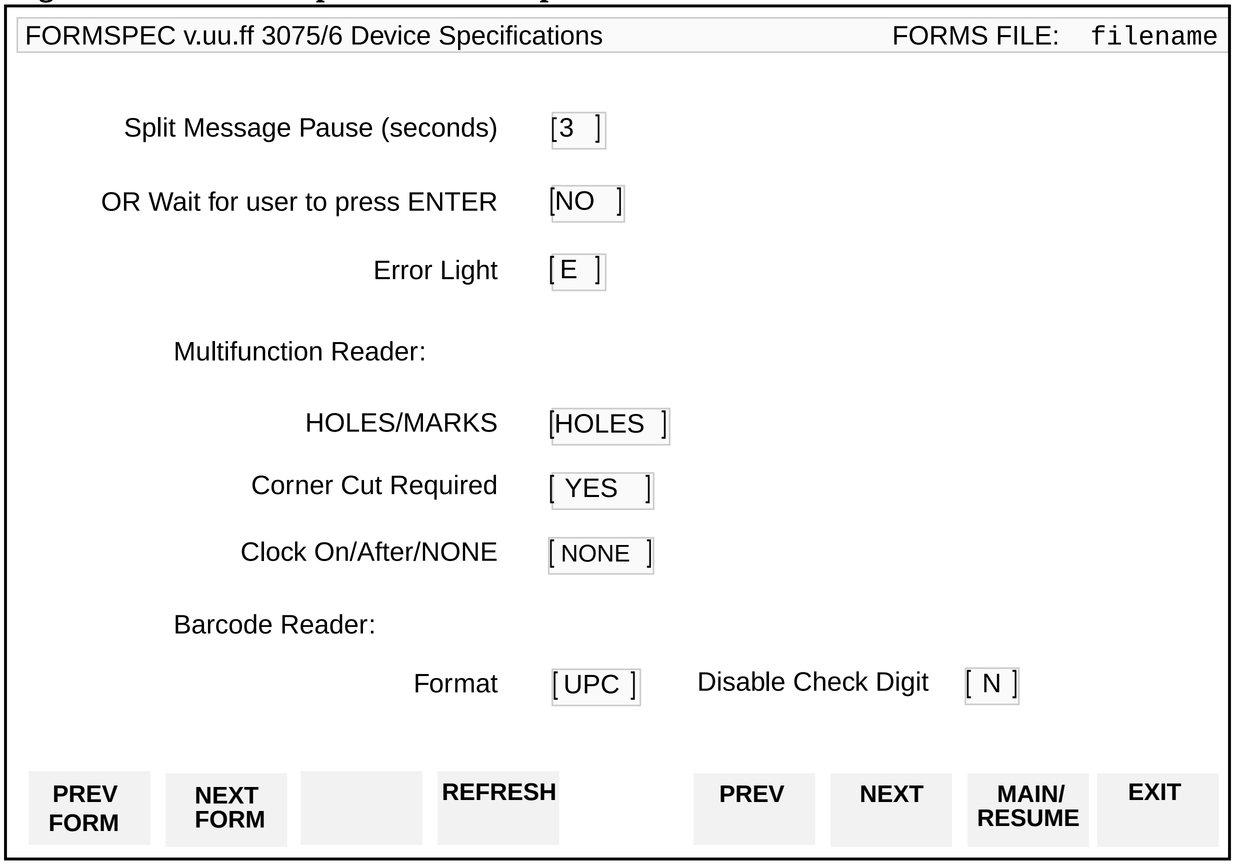
DATA CAPTURE DEVICE SPECIFICATIONS MENU

If you select the data capture devices on the Terminal/Language Selection Menu, the following message will appear at the bottom of the Globals Menu:

```
Press NEXT to select HP 3075/6 device specifications
```

Press NEXT to display the Data Capture (3075/6) Device Specifications Menu, as shown in Figure 3-32. On this menu you can specify how you want messages longer than 24 (or 32 for the HP 3081A) characters to be presented on single line display. In addition, you can specify which keyboard light is to be lit if an error is detected, and the device configuration for the Multifunction and Bar code readers if they were specified as valid input devices.

Figure 3-32. Data Capture Device Specifications Menu



FORMSPEC v.uu.ff 3075/6 Device Specifications FORMS FILE: filename

Split Message Pause (seconds) [3]

OR Wait for user to press ENTER [NO]

Error Light [E]

Multifunction Reader:

HOLES/MARKS [HOLES]

Corner Cut Required [YES]

Clock On/After/NONE [NONE]

Barcode Reader:

Format [UPC] Disable Check Digit [N]

Fields

`Split Message Pause (seconds)` You may specify the number of seconds between screen presentations for messages longer than 24 (or 32) characters. Enter a value between 1 and 99 in this box and a NO in the `Wait for ENTER` box. User and VPLUS's messages will be broken so as not to display part of a word. This field does not apply to Mini-CRT display.

Default 3 (seconds)

`OR Wait for user to press ENTER` You may also specify that screen presentations for messages longer than 24 (or 32) characters be displayed after **ENTER** is pressed. Enter a YES in this box and a zero (`0`) in the `Split Message Pause` box. This field does not apply to Mini-CRT display.

Default NO

`Error Light` You may specify which keyboard light is to be lit if an error is detected on the form by the application or by VPLUS. This light will remain lit during the re-presentation of the form. You may specify any one of the following: @, A through P.

Default E

`Multifunction Reader:` If you select the Multifunction Reader as an input device, you can specify how it should be configured to accept valid card and badge input. The following options are available:

`HOLES/MARKS` The card or badge will have holes or will be mark sensed.

Default HOLES

`Corner Cut Required` The card or badge must be presented with the corner in accordance with the diagram on the terminal.

Default YES

`Clock On/After /NONE` The card or badge will have clock marks after each column of data, or no clock marks. Valid input is CAD (clock after data) or NONE (no clocking).

Default NONE

Invalid combinations of the above are Holes with Clock On data and Marks with NONE.

`Bar Code Reader` If you select the Bar Code Reader as an input device, you can specify how it should be configured. The following options are available:

`Format` You can specify which bar code format will be read. Choose one of the following:

- `UPC` — Universal Product Code
- `I25` — Industrial 2 out of 5
- `MAT` — INTERMEC 2 out of 5 MATRIX code
- `139` — INTERMEC code 39 alphanumeric bar code
- `EAN` — European Article Numbering
- `ILV` — Interleaved 2 out of 5

Default UPC

code I39 is a registered trademark of Interface Mechanism, Inc.

`Disable Check Digit` When you specify the bar code reader as an input device, VPLUS configures the device to require check digits. You can change this by specifying yes (Y) in this field.

Default N

Example of how a typical form layout will took displayed on the data capture devices:

Figure 3-33. Form Layout in FORMSPEC

```
This is the first line of the form.

    This is line 3 of the form and it extends all the way to column 80.....

              This line starts in column 17.

                     This line starts in column 25.

                              This line starts in column 33.
```

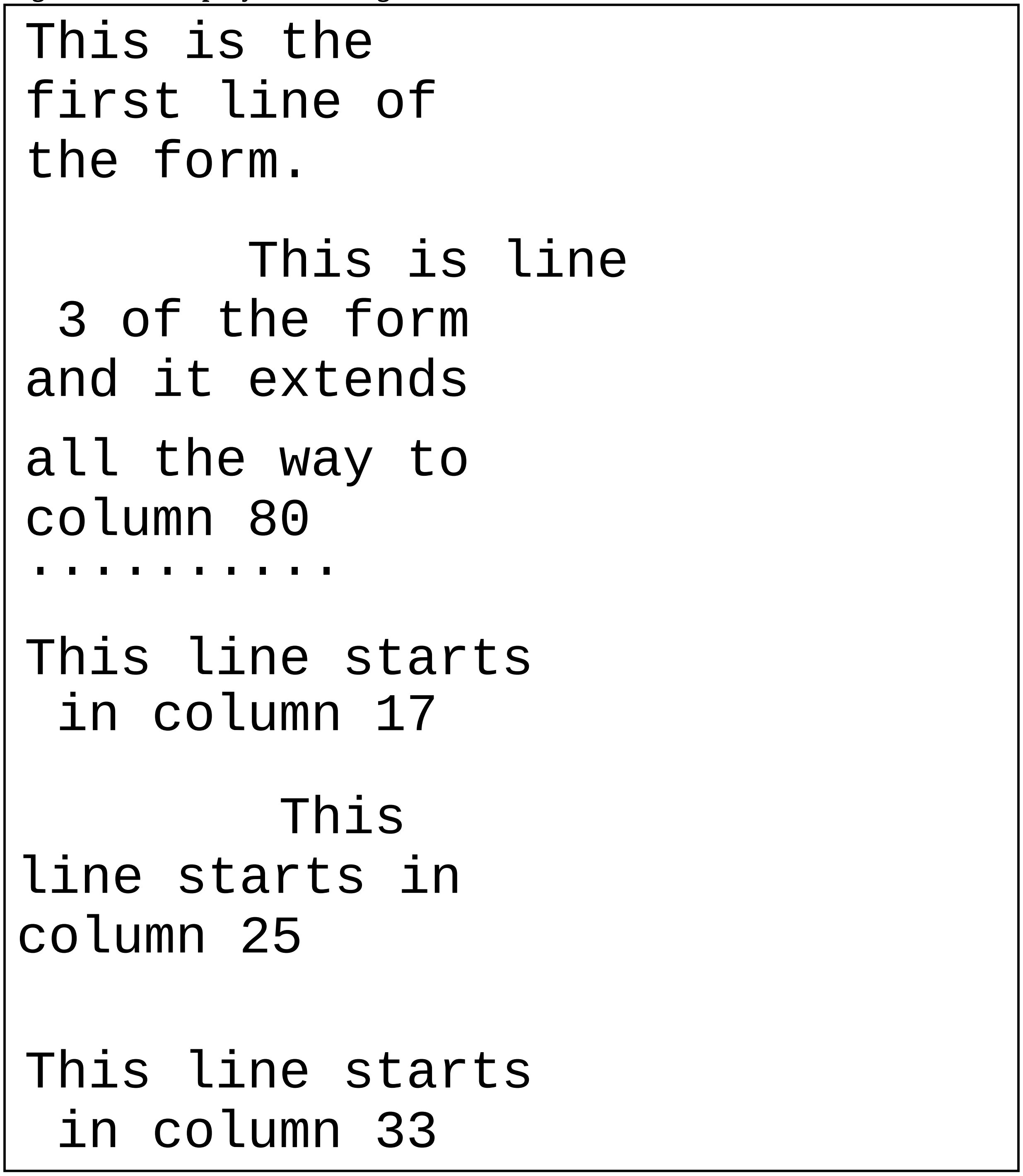
Figure 3-34. Displayed on Standard Character Set Mini-CRT Screen

```
This is the first line of the
form.
    This is line 3 of the form
and it extends all the way to
column 80..........

                This line starts
in column 17
                       This
line starts in column 25.

This line starts in column 33.
```

Figure 3-35. Displayed on 32 Character One Line Alpha Display Screen

```
This is the first line
of the form.
    This is line 3 of
the form and it extends
all the way to column 80
..........
                  This
line starts in column 17
This line starts in
column 25
         This line starts
in column 33
```

Figure 3-36. Displayed on Large Character Mini-CRT Screen



```
This is the
first line of
the form.

       This is line
 3 of the form
and it extends

all the way to
column 80
..........

This line starts
 in column 17

        This
line starts in
column 25


This line starts
 in column 33
```

4 Advanced Forms Design

The processing specifications for advanced forms design allow you to:

- Compare a field value to a specified single value, range of values, list of values, or a pattern.
- Test a check digit.
- Specify up to 20 global save fields.
- Assign a value to any field in the form, including save fields.
- Move values between fields in a form or, using save fields, between fields in different forms.
- Perform arithmetic calculations on data in numeric fields.
- Format data in a field either during data movement, or using the set of formatting commands.
- Dynamically alter the forms sequencing originally assigned to a form.
- Execute any of the processing specifications conditionally, depending on the value of a field.
- Perform processing specifications in different phases of form execution.

Levels Of Advanced Design

Advanced forms design can be separated into two levels of processing specifications:

- **Field Edits** — use a comprehensive set of editing statements that apply to a single field. These statements allow you to test a value entered in a field for length, and to compare the value to a single value, a range of values, the values in a table or against a pattern, and to test a check digit in a field. Each of these edits applies to the field in the Field Menu in which it appears — the editing statements do not cross field boundaries.
- **Advanced Processing** — includes statements that control data movement between fields and across forms, arithmetic calculation and formatting of data, dynamic alteration of forms sequencing options. Any of these statements and the edit statements can be combined into conditional statements so that processing is performed only in specified circumstances. This type of processing, as in any programming language, is affected by the order in which the statements are specified. The order of statement execution can be defined explicitly through phase statements.

Advanced forms design uses processing specifications that are specified in the lower nonformatted area of the Field Menu. Although they apply primarily to particular fields, some specifications apply to the form in general. The specifications can, if desired, be executed in four phases: configuration, initialization, field edits, and finish. If used, phases allow the selection of:

- terminal configuration for local edit terminals and data capture devices,
- initialization of fields in the form before data is entered in the fields,
- editing of data in each field after the data is entered, and,
- after all fields are edited, any finish processing of the form.

The application requests execution of each phase with appropriate VPLUS intrinsics, except for configuration, which is a passive phase built into the presentation of the form. Refer to "Phases" later in this section for more information.

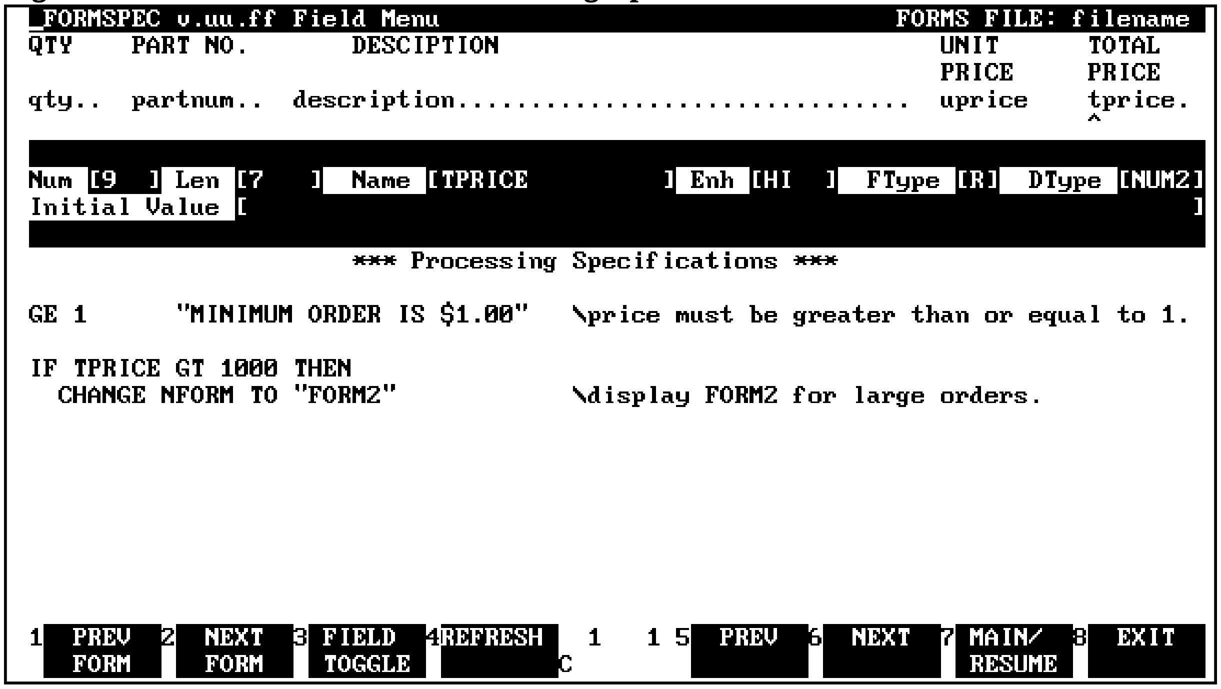
If you have collected data in a batch file, you may also use the Reformatting Capability (described in Section 5) to reformat the data in the batch file for subsequent use by an application.

Entering Processing Specifications

After defining the field attributes on the Field Menu (as described in Section 3), if you want to define additional processing, press **FIELD TOGGLE**. You can then enter the processing statements in the lower (nonformatted) area of the Field Menu. (Refer to Figure 4-1. for an illustration of the Field Menu that includes processing specifications.) When additional field processing is requested, FORMSPEC performs the following steps:

1. Leaves the Field Menu on the screen with its current values for the field attributes.
2. Places the terminal in nonformatted block mode. This allows you to type anywhere on the screen, not just in unprotected fields, and use the full terminal capabilities for block mode entry (refer to your terminal manual for details.)
3. Positions the cursor at the beginning of the first line on which you can enter the processing specifications.

Figure 4-1. Field Menu with Processing Specifications



```
_FORMSPEC v.uu.ff Field Menu                          FORMS FILE: filename
QTY    PART NO.        DESCIPTION                       UNIT      TOTAL
                                                        PRICE     PRICE
qty..  partnum..  description.............................  uprice    tprice.
                                                                  ^

Num [9  ] Len [7   ]  Name [TPRICE          ] Enh [HI  ]  FType [R]  DType [NUM2]
Initial Value [                                                                 ]

                      *** Processing Specifications ***

GE 1       "MINIMUM ORDER IS $1.00"     \price must be greater than or equal to 1.

IF TPRICE GT 1000 THEN
  CHANGE NFORM TO "FORM2"               \display FORM2 for large orders.
```

You can then enter any of the processing statements described in this section. They can be typed on any line in the processing specifications area, and must conform to the specified syntax rules. When you have typed all the specifications for this field, press **ENTER**. When **ENTER** is pressed, the field attributes are cheeked first. If there are no errors, the processing specifications are checked for syntax errors. When the Field Menu passes these checks, the next menu is displayed. You can continue with field definitions or go back to change previously defined fields.

Special Cases

If a processing specification statement line extends to the 80th column, the line is expanded to two lines when you return to the Field Menu. This may in turn result in processing specification truncation due to either terminal memory overflow or FORMSPEC internal buffer overflow.

In nonformatted mode, it is possible to accidentally clear or delete any field attributes with a key such as `CLEAR DISPLAY` or `DELETE LINE`. Doing this is only a problem if the values for the field attributes have not yet been recorded by pressing **ENTER**. To recover, press REFRESH and then reenter any changed values for the field attributes.

While in the processing specification area of the Field Menu, recovering from accidently pressing **BREAK** or from system problems requires a special recovery procedure. Refer to Appendix G.

Correcting Existing Specifications

Whenever you return to a Field Menu, the terminal is in formatted mode with the cursor positioned to the first unprotected field in the upper part of the screen. To change the field attributes, **TAB** to the field you want to change and type in the new value. To change the processing specifications in the lower part of the screen, you must press **FIELD TOGGLE** to put the terminal in nonformatted mode. The cursor is then positioned to the beginning of the processing specification area.

If you want to change a field attribute in the upper part of the Field Menu when you are in the lower part entering processing specifications, you must press **FIELD TOGGLE** to return to formatted mode. The cursor is then positioned at the first unprotected field in the upper part of the menu, and you can then change the field attributes.

Statement Syntax

The processing specifications consist of statement names followed by parameters. The description of each statement uses the conventions defined at the beginning of the manual. In order to understand the notation used in the statement formats in this section, you should review these conventions.

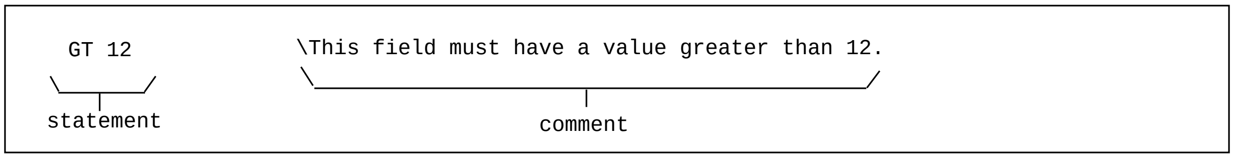
Multiple statements can be placed on the same line by following the last parameter of the statement by a blank and the next statement. If you want, you can separate statements on the same line with an optional semicolon (;). For example:

```
MINLEN 5  GT 1000  LT 2500  3 statements without separator
MINLEN 5; GT 1000; LT 2500  3 statements separated by ;
```

Statements may begin anywhere on the line, except for nested IF statements where indentation is significant. Multiple blanks are ignored within a line, except at the beginning of a line in a nested `IF` statement.

Comments

Comments may be included in the text by preceding the comment with a backslash (\). Anything typed between the backslash and the end of the line is ignored. For example:



```
GT 12            \This field must have a value greater than 12.
```

Continuing Lines

Statements that are not completed before the first backslash or the end of the line can be continued anywhere on the next line. A continuation character, the ampersand (&), is used only when a string literal must be continued on the next line. The ampersand concatenates two or more string literals to form one string. For example:

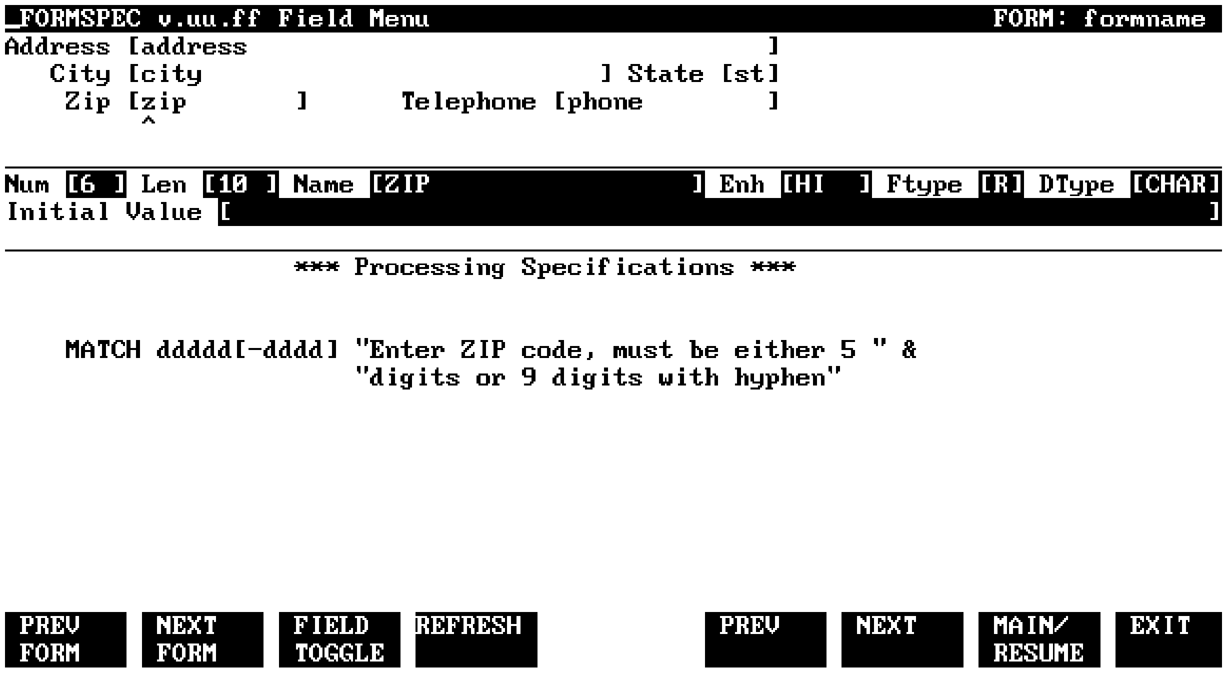
```
EQ "ABCDEF"&               \This field must be the string of
"UVWXYZ"                    \the uppercase letters ABCDEFUVWXYZ.
```

Custom Error Messages

Whenever a field edit statement detects an error at run-time, the application can call the appropriate VPLUS intrinsics to have an error message (provided by VPLUS) issued and displayed in the window line, such as is done with ENTRY. You may choose to write a custom message to be issued when a field fails a particular edit specification. To do this, you specify the custom message in quotes immediately following the statement to which it applies. Figure 4-1. illustrates a Field Menu in which the processing specifications contain custom error messages. When and if the statement causes a field to fail, the custom message is displayed instead of a VPLUS error message. For example:

```
MATCH udddd       "Field has wrong format for Product Number"
```

Figure 4-2. Field Menu with Custom Error Messages



```
_FORMSPEC v.uu.ff Field Menu                                      FORM: formname
Address [address                                   ]
   City [city                           ] State [st]
    Zip [zip       ]       Telephone [phone        ]
         ^

Num [6 ] Len [10 ] Name [ZIP                   ] Enh [HI  ] Ftype [R] DType [CHAR]
 Initial Value [                                                                 ]

                   *** Processing Specifications ***


    MATCH ddddd[-dddd] "Enter ZIP code, must be either 5 " &
                       "digits or 9 digits with hyphen"
```

Checking Data Entered

The field attributes for the field determine the first checks made on the data in the field. For example, if the field is required (FType=R), it is checked to be sure that the user has entered a value in the field. Editing is performed according to the data type. For example, if the data type is DIG, then the field is checked to be sure that only the digits 0-9 were entered in the field. Next, the processing specifications are tested in the order they are specified. As an example, consider the specifications in Figure 4-2. After making the checks based on the field attributes, data entered in the field is tested to be sure that at least 5 digits were entered (MATCH ddddd), and optionally the value is checked for a hyphen and 4 more digits ([- dddd]). In all cases, the checks are performed by VPLUS when the application calls the appropriate VPLUS intrinsics, as described in Section 6.

The following sections provide a summary of all the statements that may be entered in the Processing Specifications area of the Field Menu. Each of these statements is fully described later in this section.

CONFIGuration Statements

```
LOCALEDITS [ALPHABETIC
            ALPHANUMERIC
            CONSTANT
            IMP_DEC
            IMP_DEC_FILL
            INTEGER
            INTEGER_FILL
            SIGN_DEC
            SIGN_DEC_FILL
            UNRESTRICTED
            UPSHIFT       ] [JUSTIFY
                             MUST_FILL
                             REQUIRED] ...[DEC_DIGITS n]
                            [DEC_TYPE_EUR
                             DEC_TYPE_US  ]
                                            [TRANSMIT_ONLY]
```

```
LIGHT {@
       A
       ...
       O
       P   }...
```

```
DEVICE {PRINTER
        KEYBOARD
        MAGSTRIPE
        TYPEV
        MULTIFUNCTION
        READER
        MFR [CADMARKS
             NONE HOLES
             CAD HOLES [CUT
                        NOCUT]]
        DISPLAY [LARGECHAR
                 STDCHAR]
        BARCODE [READER] [UPC
                          I25
                          I39
                          MAT
                          EAN
                          IVL]}...
```

Edit Statements

```
  MINLEN value ["msg"]              {GT
                                     LT
                                     GE
                                     LE
                                     EQ
                                     NE}        value ["msg"]
{IN
 NIN} { value[,value]
        lowvalue:highvalue[,lowvalue:highvalue]...
        value[,lowvalue:highvalue]...                   } ["msg"]

MATCH pattern ["msg"]                         CDIGIT {10
                                                      11} ["msg"]
```

Formatting Statements

```
STRIP  { TRAILING
         LEADING
         ALL      } "character"             JUSTIFY { LEFT
                                                      RIGHT
                                                      CENTER}
FILL   {LEADING
        TRAILING }  "character"              UPSHIFT
```

SET Statements

```
SET { TO     { value
               index OF value[,value]...}
      {field
       save field} [TO { value
                         index OF value[,value]... } ] }
```

CHANGE Statements

```
CHANGE {CFORM TO {NOREPEAT
                  REPEAT
                  REPEAT APPEND}
        NFORM TO {[CLEAR
                   APPEND
                   FREEZE APPEND] [ "formname"
                                    fieldname
                                    index OF value[,value]...
                                    $RETURN
                                    $HEAD
                                    $END
                                    $REFRESH]}}
```

Conditional Statement

```
IF [ value] editstatement THEN [statement]...
    [statement]...

[ELSE [statement]...
      [statement]...
```

Control Statement

```
FAIL [“msg”]           STOP
```

Operands

Whenever *value* is specified as an operand in a field processing statement, a field name, constant, expression, or a save field name can be used. If the operand is numeric, any of these can be combined into an arithmetic expression whose result is used as the operand.

Another operand, *index*, is used to retrieve items from a list by an index value. It must be enclosed within parentheses in edit statements; in SET and CHANGE statements, it can be specified without the enclosing parentheses.

Field

The name of any existing field within the same form can be specified as a *value* operand. When a field name is specified, the value in that field at run-time is used to check the current field.

A field name in an editing statement must describe a field of the same data type (character, numeric, or date) as the current field.

Constant

Constants are divided into categories corresponding to the three main data types: character, numeric and date. In addition, there are four system defined constants:

$EMPTY	equivalent to all blanks; in comparison, $EMPTY is less than any other value (*any data type*).
$LENGTH	equivalent to the length of the field the last time the form was compiled (*numeric data types*).
$STATE	equivalent to a table of constants each of which is a two-character upper or lowercase United States state or territory code; refer to Appendix F for the codes (*character type; used only in table checks*).
$TODAY	equivalent to today's date in the form *dd/dd/dd* where the order of month day year is determined by the field type (*date type only*).

Character Character constants are strings of any character enclosed in single or double quotes.

For example:

```
    "This is a constant."
    'and this is, too!'
    $EMPTY
    $STATE
```

Numeric Numeric constants are digit strings with an optional leading sign and optional decimal point. Commas are not allowed in numeric constants. If the constant has no decimal point, a decimal is assumed at the right of the rightmost (least significant) digit. $EMPTY (all blanks) can be used to indicate a numeric constant less than any non-$EMPTY value, $LENGTH can be used to indicate the length of the current field. The following examples Illustrate numeric constants:

```
    .23
    -123
    5000
    -3729
    $EMPTY
    $LENGTH
```

Date. A date constant can be any legal date format, but it must be in the order MDY. With Native Language Support, the date constant order (e.g., *mmddyy*) will not be language dependent. For more information on Native Language Support, see Section 8. To distinguish it from other constants, a date constant must be delimited by exclamation points (!). Note that the date constant in the order MDY is used to check date values in any of the three legal orders: MDY, DMY, or YMD. A special date constant $TODAY is equivalent to the date at execution time. Examples of date constants are:

```
!MAY 30, 1986!
!12/24/86!
!NOVEMBER 7, 86!
$TODAY
$EMPTY
```

Arithmetic Expression

Any numeric constant, field, or save field can be combined into an arithmetic expression. The expression is evaluated to generate a constant that can be used to check a field or replace the value of a field. The operators used to form an arithmetic expression are:

+	(add)
-	(subtract)
*	(multiply)
/	(divide)
%	(percent of)

The operators determine the order of evaluation in the standard operator hierarchy where + and - are evaluated after *, /, and %. Operators at the same level are evaluated from left to right. Parentheses may be used to further define the hierarchy. Expressions within parentheses are evaluated first and, if parentheses are nested, the innermost are evaluated first. The % in the expression a%b is equivalent to (a*.01)*b. When $EMPTY is used in an arithmetic expression, the result is always $EMPTY. Thus, $EMPTY with any operator and operand = $EMPTY. The following examples illustrate arithmetic expressions:

`F2 + 1`	Add 1 to the value in field F2.
`QUANTITY * UNITPRICE`	Multiply value in QUANTITY by value in UNITPRICE.
`TOTALCENTS/ 1000`	Divide value in TOTALCENTS by 1,000.
`20% (TOTALPRICE - TAX)`	Find 20 percent of the value resulting from subtracting TAX from TOTALPRICE.
`$EMPTY + 20`	The result is always $EMPTY, regardless of the operator or other operand.

NOTE When the VPLUS intrinsics execute an arithmetic expression, they expect each operand in the expression to have a value; if any operand does not have a value, the result is null. It appears to the user as if the expression had not been executed and no message is issued.

Index Retrieve Operand

An index retrieve operand (enclosed within parentheses) can be used in any edit statement where a constant, field, or expression is legal. An index retrieve operand without enclosing parentheses can be used in SET and CHANGE statements.

An index retrieve operand is specified in the format:

index OF *element [,element]*...

index	is a numeric type field or save field, or a numeric expression whose value is a positive integer. The index value indicates which element is selected.
element	is a constant, field, save field, or expression. Each element in the list of elements separated by commas must be the same type (character, numeric, or date).

When an operand of this type is specified, its effective value is selected from a list of values according to its position in the list, where the first element of the list is 1. First, the index expression is evaluated, then this value is used to select a final value from the list.

For example:

```
EQ (N OF "ABC","DEF","GHI", FIELDX)
```

If the value of N is 1, the character string "ABC" is the selected operand: if the value of N is 4, the value of FIELDX is selected. If the value of N is negative, zero, or greater than the number of elements, an error message is returned. Note that the operand is enclosed in parentheses because EQ is an edit statement.

```
SET TO F3 OF F1,15,20, 25,35,40
```

If the value of F3 is 1, the selected operand is the value of field F1; if F3 is

4, the selected operand is 25. Parentheses are not required in a SET statement.

```
CHANGE NFORM TO X OF
"FORMA", "FORMB", "FORMC",
OPSELECT
```

X may be a value between 1 and 4. If its value is 4, the field name OPSELECT is chosen. The value of OPSELECT must be type character to conform to the other elements in the list, and to make sense it should contain a form name. No parentheses are needed in a CHANGE statement.

If an index has a value beyond the number of elements in the list or is $EMPTY, an error is returned.

Processing Specifications For Configuration

You can use the processing specifications for configuration to specify local editing to be done on the HP 2394A and HP 2624A/B terminal, and to specify the input device and prompting lights for use on the data capture devices.

These statements may only be used in the configuration phase of field processing, and must have the `CONFIG` heading in the processing specification area of the Field Menu.

NOTE VPLUS edit processing specifications and terminal edit processing statements are separate and are not checked for compatibility. There will be no check to see that the designer has specified a terminal local edit (`DEC_TYPE_EUR, DEC_TYPE_US`) in the configuration phase which is consistent with the language-dependent decimal indicator for the native language specified in the forms file. For more information on Native Language Support see Section 8.

DEVICE

Enables the input media device on a data capture device.

Syntax

DEVICE {*device1,device2,...devicen*}

Parameters

device	Any of the following may be specified:

```
PRINTER
KEYBOARD
MAGSTRIPE
TYPEV
MULTIFUNCTION READER
DISPLAY
BARCODE READER
```

There are no default devices. Note that `MULTIFUNCTION READER` and `BARCODE READER` must be abbreviated as `MFR` and `BARCODE` respectively.

Discussion

This command allows you to override the global attribute values for the Multifunction Reader and Bar Code Reader. You must specify which devices are to be used for input/output.

NOTE The Interleaved 2 out of 5 bar code is only supported from the Globals Menu. If you wish to use this bar code format, it must be defined globally from that menu.

Example

```
                  ***Processing Specifications***

CONFIG
DEVICE PRINTER, KEYBOARD, DISPLAY, BARCODE READER
```

LIGHT

Specifies which light will be lit during field presentation on a data capture device.

Syntax

LIGHT *{light1,light2,...lightn}*

Parameters

light	Specifying @ turns on the light associated with the shift key. The remaining valid characters of A through P are associated with the same characters on the alphanumeric keyboard.

Discussion

This command allows you to specify which, if any, of the seventeen available prompting lights is to be turned on during a field presentation. As many lights as desired can be specified.

Example

```
                      ***Processing Specifications***

     CONFIG
     LIGHT A,B,D,G,N,P
     LIGHT @
```

LOCALEDITS

Used to stipulate edits to be done within the terminal.

Syntax

```
LOCALEDITS {edit1,edit2,...editn}
```

Parameters

edit	Table 4-2. lists available local edits along with their meanings. Refer to Table 4-1. for valid local edit combinations. For example, you can specify alphanumeric, required, decimal type, and number of digits to the right of the decimal together. You cannot specify alphanumeric and upshift together.

Discussion

This command allows you to specify local edits that are performed as keys are pressed by the user entering data into the form. At run-time, VPLUS checks the terminal type and, if the terminal supports local edits, loads the specified local edits to the terminal. If the terminal does not support local edits, VPLUS ignores them.

The last three decimal edits in Table 4-2. alter the mode of the terminal; therefore, the last field for which a decimal edit is specified that is displayed on the terminal alters the way the terminal processes all fields until the terminal mode is changed by another screen containing a decimal edit. All other local edit specifications affect only the field for which they are specified.

When an invalid key is pressed, the terminal beeps, and an error message issued by the terminal (not by VPLUS) is displayed. Press **RETURN** to clear, then reenter data into the field.

If you specify local edits with a form family, the local edits associated with the first displayed form in the family are retained as subsequent family members are displayed (in other words, it is not possible to change local edit specifications among family members at run-time).

VPLUS "edit" processing specifications and "terminal" edit processing statements are separate and are not checked for compatibility. There will be no check to see that the designer has specified a terminal local edit (DEC_TYPE_EUR, DEC_TYPE_US) in the configuration phase which is consistent with the language-dependent decimal indicator for the native language specified in the forms file. For more information on Native Language Support, see Section 8.

Table 4-1. Summary of Processing Statements

Local Edit	Description
`ALPHABETIC`	Upper and lowercase alphabetic, space, period, hyphen
`ALPHANUMERIC`	Upper and lowercase alphabetic, digits, space, period, minus sign, plus sign, comma.
`CONSTANT`	No characters may be entered from the keyboard,
`IMP_DEC`	Implied decimal (the number of digits to the right of the decimal is governed by the last `DEC_DIGITS` command in the form).
`IMP_DEC_FILL`	Implied decimal, right justified and zero filled (the number of digits to the right of the decimal is governed by the last `DEC_DIGITS` command).
`INTEGER`	Digits and spaces.
`INTEGER_FILL`	Digits and spaces, right justified and zero filled.
`SIGN_DEC`	Signed decimal.
`SIGN_DEC_FILL`	Signed decimal, right justified and zero filled.
`TRANSMIT_ONLY`	No data may be entered unless the cursor control keys are used to access the field.
`UNRESTRICTED`	Any character.
`UPSHIFT`	Lowercase alphabetic changed to uppercase when entered.
`JUSTIFY`	right justify numeric types; left justify character types.
`MUST_FILL`	The field must be blank or filled entirely.
`REQUIRED`	The field must have an entry when **ENTER** is pressed.
`DEC_TYPE_EUR`	European number format.
`DEC_TYPE_US`	United States number format.
`DEC_DIGITS` *n*	Specifies number of digits to right of decimal point.

Here is a table showing the local edit terminal menu selection and associated VPLUS local edits.

Table 4-2. Local Edit Terminal Menu Selection

Local Edit	Terminal Menu Selection	Justify is used
UNRESTRICTED	0. ALL CHARACTERS	Left Justify
ALPHABETIC	1. ALPHABETIC	Left Justify
UPSHIFT	2. AUTO UPSHIFT	Left Justify
ALPHANUMERIC	3. ALPHANUMERIC	Left Justify
INTEGER	4. INTEGER	right justify
SIGN_DEC	5. SIGNED DECIMAL	right justify
IMP_DEC	6. IMPLIED DECIMAL	Ignore
CONSTANT	7. CONSTANT	Ignore
INTEGER_FILL	8. INTEGER/FILL	Ignore
SIGN DEC FILL	9. SIGNED DECIMAL/FILL	Ignore
IMP_DEC_FILL	10. IMPLIED DECIMAL/FILL	Ignore

Table 4-3. Local Edit Terminal Menu Selection with Defaults

Local Edit	Terminal Menu Selection	Default
REQUIRED	REQUIRED	OPTIONAL
JUSTIFY	JUSTIFY	NO JUSTIFY
MUST_FILL	TOTAL FILL	NO TOTAL FILL

For each field, the user may select one edit from Table 4-2. and any combination of edits from Table 4-3. according to the following guidelines:

1. If no selection is made from Table 4-2., the default edit of UNRESTRICTED is used.
2. The CONSTANT and REQUIRED edits are incompatible, since the required data cannot be entered in the CONSTANT field.
3. The terminal executes all edits implied by field type before it checks for TOTAL_FILL. The MUST_FILL edit is always met by INTEGER_FILL,SIGN_DEC_FILL, and IMP_DEC_ FILL.
4. The effect of the JUSTIFY attribute on each of the field types is shown in Table 4-2.

Table 4-4. Local Edit Terminal Menu Items

Local Edit	Terminal Menu Item	Menu Selection
DEC_DIGITS *n*	Implied Dec digits	0-9
DEC_TYPE_EUR	Decimal Type	EUR
DEC_TYPE_US	Decimal Type	US

Along with the selections from Table 4-2. and Table 4-3. you may specify the number of decimal digits (DEC_DIGITS *n*) and select a decimal format (DEC_TYPE_US or DEC_TYPE_EUR). (Note that the DEC_DIGITS *n* setting may only be used with IMP_DEC or IMP_DEC_FILL edits.) Unlike other edits, DEC_DIGITS and DEC_TYPE edits are global terminal configurations that remain in effect until they are reset by another DEC_TYPE or DEC_DIGIT edit. Please note that VOPENTERM and VCLOSETERM do not reconfigure DEC_DIGIT and DEC_TYPE; these settings may carry over from one application to the next.

Example

```
                  ***Processing Specifications***

CONFIG                                 \No characters may be entered
LOCALEDITS CONSTANT                    \from the keyboard.
CONFIG                                 \Entered data must be in United States
LOCALEDITS DEC_TYPE_US,REQUIRED        \number format; the field must contain
                                      \data when ENTER is pressed.
```

Processing Specifications For Field Edits

If only field editing is needed, the edit statements listed below provide full field edits. Edit statements apply to the current field only. During table and range checks leading and trailing blanks are stripped. (The current field is the field defined in the Field Menu in which the edit statements are specified.)

Edit Statements

The FORMSPEC edit statements allow you to check whether the value of the current field:

- is at least a specified minimum length; refer to "Length Check" for the `MINLEN` command.
- is greater than, equal to, or less than a specified value; refer to "Single Value Comparisons" for the `GT, GE, LT, LE, EQ`, and `NE` commands.
- is one of a list (table) of values, or is not in the list; refer to "Table Checks" for the `IN` and `NIN` commands.
- is within a range of specified values; refer to "Range Checks" for the `IN` and `NIN` commands.
- has a valid check digit; refer to "Check Digit" for the `CDIGIT` command.
- matches a general pattern of characters; refer to "Pattern Match" for the `MATCH` command.

Edit statements use the set of statements listed in Table 4-1. They are performed in the order they are entered by the forms designer. As soon as an edit fails during execution, the field is marked in error and further processing on that field is abandoned.

LENGTH CHECK

Used to specify the minimum length of a field value.

Syntax

MINLEN *value ["message"]*

Parameters

value	The value can be a field, constant, save field, or a numeric expression, or it may be an index retrieve operand within parentheses. Refer to "Statement Syntax" earlier in this section for details.
message	Specifies a custom error message.

Discussion

A value longer than the field cannot be entered because of the physical limit imposed by the unprotected field area. For this reason, maximum field length need never be specified as an edit check. You may, however, specify the minimum number of characters to be entered with the MINLEN statement.

The value specifies the minimum number of characters allowed in the field. Note that this length does not include leading and trailing blanks.

The system defined value $LENGTH may be specified. This value allows you to require that the field be filled. The advantage of $LENGTH rather than the current field length is that $LENGTH allows you to change the field length without changing the MINLEN specification. The $LENGTH constant is equal to the length of the field when the forms file is compiled.

Example

1. For example, the minimum number of characters entered in a six-character field could be specified as:

   ```
   MINLEN 2
   ```

   If the user spaces over three characters, types in one nonblank character, and leaves the rest of the field blank, an error is diagnosed.

2. In another example, MINLEN is used to specify that the field must be filled. The following statement forces the field to filled, leaving no leading or trailing blanks:

   ```
   MINLEN $LENGTH
   ```

   You can also use an indexed operand as follows:

   ```
   MINLEN (LEN OF F1,F2,F3,F4,F5)
   ```

LEN must contain an integer value between 1 and 5, and the current value of the fields F1 through F5 must be positive. The minimum length depends on the value of LEN and the respective values of F1 through F5.

SINGLE VALUE COMPARISONS

Used to compare the field value against a single value.

Syntax

```
[{GT
  LT
  GE
  LE
  EQ
  NE}    value    ["message"]]
```

Parameters

value	The value can be a field, constant, save field, or a numeric expression, or it may be an index retrieve operand within parentheses. Refer to "Statement Syntax" earlier in this section for details.
message	Specifies a custom error message.

Discussion

You may use these statements to compare the contents of the current field with a specified value. Any field can be compared to the special comparison value $EMPTY that represents the lowest value for any data type. The main rule governing these comparisons is that only values of the same data type (character, numeric, or date) may be compared. The comparison is performed from left to right after leading and trailing blanks have been stripped from the field.

Character Comparisons

If the two values being compared are not the same length, the shorter is padded with blanks on the right until it is the same length as the longer value. Then a comparison is made.

In a character comparison, values are considered equal only if each character matches. One value is less than another if, at the point of mismatch, one character is numerically less than its counterpart. Collating sequence order is used to determine the numeric value of a character. (Refer to Appendix C for the USASCII collating sequence; refer to the *Language Support Reference Manual* for the collating sequences for all supported native languages.) For example, for NATIVE-3000:

a	is greater than A
A	is greater than Δ
AA	is greater than A
AΔ	is greater than Δa

where Δ represents a blank.

Numeric Comparisons

When numeric values are compared, they are first converted to the HP 3000 internal representation of the number. Thus, a field of type `IMPn` can be compared with `NUMn`, and `NUMn` or `IMPn` fields can be compared to type DIG fields. For example, assume a field of type `NUM` 2 in which the value `123` has been entered. This value can be successfully compared to the constant "123", or a field of type `IMP, NUM`, or `DIG` so long as the field contains the value `123` at run-time. For example:

```
123.000 (NUM3) is equal to 123 (DIG)
123.000 (NUM3) is equal to 12300 (IMP2)
```

Date Comparisons

Two dates may be compared even if they differ in format and order. This means that the date `February 9`, 1986, specified in the current field as `9 FEB 1986` (DMY) can be successfully matched against the constant ! `02/09/86`! whose order is MDY. Remember: a date constant must always be in the order MDY. For example:

```
2/3/86 (MDY) is equal to Feb 3, 1986 (MDY)
2/3/86 (MDY) is equal to 86/2/3 (YMD)
```

Example

`EQ F3`	The current field value must exactly match the value in `F3` at run time. (Any leading or trailing blanks are stripped from the value in both fields before the comparison is made.)
`GT 143.56`	The current field value must be greater than `143.56`
`NE $TODAY`	A date entered in this field cannot be today's date.
`GT "Cd"`	The current field may contain any value greater than `Cd`, such as `D` or `Cde` but it cannot be `C` or `CD`. Note that in NATIVE-3000 all uppercase characters have a lower value than any lowercase character.
`GE !5/7/86!`	Any date including and after `May 7, 1986` may be in current field. The date may be in any of three formats: MDY, DMY, or YMD.

Native Language Support

NATIVE-3000 must be used to define values for date and numeric fields within FORMSPEC. VPLUS will convert the value when the forms file is executed to be consistent with the native language selected. Single value comparisons (`GT, GE, LT, LE, EQ, NE`), specified within FORMSPEC may contain any character in the 8-bit extended character set consistent with the selected language ID. When the form is executed, the collating table for the native language specified is used. For more information on Native Language Support, see Section 8.

TABLE CHECKS

Used to verify that the field value is either in a list (table) of values, or is not in that list.

Syntax

```
[{IN
  NIN} value1[,value2]...[“message”]]
```

Parameters

value	The values specified in the list may be any mixture of field names, save fields, constants, numeric expressions, or it may be an index retrieve operand within parentheses. Refer to "Statement Syntax" earlier in this section for details.
message	Specifies a custom error message.

Discussion

Each element in the list must result in a single value to be matched exactly, except for leading and trailing blanks in the field. There is an implicit *OR* between the elements of the list so that the statement can be understood as:

Is current value equal to (IN) value1 *OR* value2 *OR*...
Is current value not equal to (NIN) value1 *NOR* value2 *NOR*...

One system defined table, $STATE, is provided that consists of a list of all two-character United States state and territory codes (see Appendix F for a list of the codes). This list is in alphabetic order and can be compared successfully with all upper or lowercase codes. The $STATE constant may be used anywhere a table is legal in the statement syntax.

As with other comparison, values must match the field data type; that is, a numeric field can only be compared to a list of numeric values, a date field to a list of date values, and a character field to a list of character values.

Example

`IN 12, 14, 16, 18, 20`	Current value must be one of the five listed numbers.
`NIN "CA", "ME", "NY"`	Current value must not be any of the three listed values.
`IN $STATE, MX`	Value must be legitimate state code or be MX. The system constant $STATE can be in a list including other values.

Native Language Support

NATIVE-3000 must be used to define values for date and numeric fields within FORMSPEC. VPLUS will convert the value when the forms file is executed to be consistent with the native language selected. Table checks specified within FORMSPEC may contain any character in the 8-bit extended character set consistent with the selected language ID. For more information on Native Language Support, see Section 8.

RANGE CHECKS

Used to check whether the field value is within, or is not within, a specified range of values.

Syntax

[{IN
NIN} *lowvalue*:*highvalue* [,*lowvalue*:*highvalue*]... ["*message*"]]

Parameters

value	The value can be a field, constant, save field, or a numeric expression, or it may be an index retrieve operand within parentheses.
message	Specifies a custom error message.

Discussion

Range checks are similar to table checks except that you specify a list of ranges rather than a list of exact values. The range is inclusive, that is the field value must be within the range that includes the lowvalue and the highvalue.

The low and high values may be any combination of field names, constants, save fields, arithmetic expressions, or index retrieve operands. As with table checks, an implicit OR is understood between the ranges in the list. Thus, the statement can be interpreted as:

```
Is current value in (IN) range1 OR range2 OR...
Is current value not within (NIN) range1 NOR range2 NOR. . .
```

The low value must not be greater than the high value. If it is, an error is issued when the form is executed (not when it is compiled).

Example

`NIN 12:45`	Current numeric value must not have any value between 12 and 45 inclusive.
`IN F2/2:F2*2`	The field can have any numeric value ranging from half the value in field F2 through twice the value in F2. Note that the current field and F2 must be numeric in order to use an arithmetic expression in the range check.
`IN $TODAY:! 12/12/99!`	Any date between today's date and December 12, 1999 may be entered.

Range and table checks can be combined in one statement, as illustrated in the following examples:

`NIN -12.5:-2,25, 1000:FIELD3`	Value may *NOT* be in range -12.5 through -2, nor equal to 25, nor in the range from 1000 through the current value of FIELD3.

`IN "ADE": "BB", "s", "t"`	The field may have any value in the range ADE through BB or it can be s or t. Thus, it could be Abcd, or B or t, but it cannot be AB or S. In NATIVE-3000, all lowercase letters are greater than any uppercase letters.
`IN "20":"30"`	Allows any character values between 20 and 30 in the USASCII collating sequence. Thus 2A and 3& are valid for NATIVE-3000. (Other native languages use other collating sequences.)

Native Language Support

NATIVE-3000 must be used to define values for date and numeric fields within FORMSPEC. VPLUS will convert the value when the forms file is executed to be consistent with the native language selected. Range checks specified within FORMSPEC may contain any character in the 8-bit extended character set consistent with the selected language ID. When the form is executed the collating table for the native language specified is used to check whether the field is within range. For more information on Native Language Support, see Section 8.

CHECK DIGIT

Used to test the check digit in a numeric or alphanumeric field (modulus 10 or 11).

Syntax

```
[CDIGIT {11
         10} ["message"]]
```

Parameters

message	Specifies a custom error message.

Discussion

Check digit verification is a special check on a numeric or alphanumeric field in which the last (rightmost) character is a check digit. Verification can be either modulus 10 or modulus 11. (You can use `REFSPEC`, as described in Section 5, to add a check digit to an existing field value in a batch file.)

Modulus checks are used when the risk of error keying in numbers must be reduced to a minimum. Depending on the modulus selected, single digit errors, and single or double transpositions can be checked using a check digit. Modulus 10 detects single transpositions and incorrect keying of a single digit. Modulus 11 detects these, plus double transpositions. (Refer to Appendix D for an exact description of the modulus 10 and 11 checks.)

In general, a check digit is arrived at by performing calculations on a number and then using the result of these calculations as the final digit or "check digit" in that number. For example, suppose a 5-digit charge account number is to be assigned to a new account. The specified calculations (modulus 10 or 11) are performed on the 5-digit number, the result is added as a check digit to the number, and a 6-digit number is assigned as the new account number.

Thereafter, when this number is keyed in and the `CDIGIT` edit test is selected, then the last digit is checked against the same calculations. If the number was keyed incorrectly, an error is diagnosed.

Note that a number derived using modulus 10 calculations can only be checked by `CDIGIT 10`, and a number derived using modulus 11 can only be checked by `CDIGIT 11`.

Letters of the alphabet can be checked by either a modulus 10 or 11 check. Digits are assigned to the letters so they can be treated like numbers. Thus a field with a check digit can contain a mix of numbers and letters, but must not contain any special characters. Any initial plus or minus sign is ignored.

PATTERN MATCH

Used to test a field value against a pattern of characters.

Syntax

```
MATCH pattern      ["message"]
```

Parameters

pattern	A pattern of characters which indicate which type of character in which position is acceptable input for this field.
message	Specifies a custom error message.

Discussion

The pattern match allows you to check field values against a general pattern. It can be used to check the actual value, but actual values are more easily checked in the Single Value, Table, or Range tests. This test is generally used to test the type of character entered in a particular position. Before data is checked against a pattern, any leading and trailing blanks are stripped.

NOTE Native Language Support does not support pattern matching with native language characters. MATCH uses USASCII (NATIVE-3000) rules. For more information on Native Language Support, see Section 8.

Special Pattern Characters

The pattern consists of a series of special characters that indicate the type of USASCII data that can be entered in that position. These characters are:

a	upper or lowercase alphabetic character (A-Z, a-z)
u	uppercase alphabetic character (A-Z)
l	lowercase alphabetic character (a-z)
b	blank
d	digit (0-9)
?	any character

The beginning of the pattern is defined by the first nonblank character after MATCH and, in the simplest case, is terminated by the first blank encountered. A pattern beginning or ending with the special character b will interpret the b as the alphabetic character and not the special character indicating a blank. Leading and trailing blanks should be indicated as spaces within a grouping as described on the next page.

A pattern may contain embedded blanks only if it is enclosed within braces {}. Also, a pattern can span more than one line if it is enclosed within braces {}. If a pattern is within braces, it is terminated by the first blank outside the braces.

The match pattern can include specific characters in addition to the types listed above. To illustrate:

```
MATCH Aaa-ddd
```

This pattern means that the value must start with the letter A, be followed by any two upper or lowercase letters of the alphabet, followed by a hyphen and then three digits. For example, the value `Acs-123` or `AAA-999` are acceptable, but the values `Bcs-999` or `A12-345` are not.

Transparency

A special operator can be used to indicate that a pattern character is to be used as an actual value. For example, suppose you want the lowercase letter a to be an exact value in the pattern. You can do this by preceding it, or any of the other special pattern characters, with an exclamation point (!). For example:

```
MATCH !addd          \Value must start with the letter "a"
                     \followed by any three digits.
```

The exclamation point (called the transparency user) is also used to allow inclusion of any of the pattern operators listed below, and described in Table 4-2.

!	(tranparency)
,	(choice)
:	(range)
{}	(grouping)
[]	(optional)
+	(repetition — 1 or more)
*	(repetition — 0 or more)

Choice

You can indicate a selection of acceptable patterns as part of the `MATCH` pattern. Each possible choice is separated by a comma (,). For example:

```
MATCH A,AB,BCD,ddd            \The characters "A", "AB", "BCD",
                              \or any three digits are acceptable.
```

Range

A range of acceptable characters for a single character position can be indicated with the colon (:). All characters within the range are acceptable. This acts as a shorthand for listing a series of single characters in USASCII sequence. For example:

`MATCH C:J`	This is equivalent to the pattern specified by `MATCH C,D,E,F,G,H,I,J`.
`MATCH 1:7`	Any of the digits 1,2,3,4,5,6, or 7 is accepted by this

	pattern.
`MATCH !a:f`	Since the letter a is a pattern character, it is preceded by an exclamation point. Other such characters within the range are implicitly preceded by this operator. Thus, the range is equivalent to specifying `MATCH !a,!b,c,!d,e,f`.

It is important to differentiate between a pattern range which is a range of single characters, and the range check described earlier. In a pattern check, `MATCH 10:90` means the value must be a `1` followed by a digit between `0` and `9`, followed a `0`. In a range check, `IN 10:90` means the value must be in the range `10` through `90`.

Grouping and Optional

You can group pattern specifications by enclosing the pattern in braces {}. Brackets [] make the enclosed pattern optional. Braces indicate data must correspond to at least one item in the group; brackets indicate any item in the group is optional. For example:

`MATCH {A,AB,BCD}) ddd`	One of the choices within braces must be matched. For example: `A123` or `AB999` or `BCD562`, among others, are acceptable matches for this pattern.
`MATCH [A,AB,BCD] ddd`	All choices within brackets can be omitted, or one may be matched. For example: `AB345, BCD567, A441`, or `123` are all acceptable matches.
`MATCH [u,d]!+ [1:5]`	Some acceptable values are `A+` or `5+3` or `+5` or simply `+`.
`MATCH [B,dd]dd[%,d]`	Accepts such values as `B12` or `12345` or `50%` or `10`, among others.

Since blanks may be included within braces, you can put blanks in a pattern to enhance its clarity by enclosing the entire pattern within braces. For example,

`MATCH {[B,dd] dd[%,d]}`	Identical to preceding pattern except that it is enclosed within braces so that blanks can be included.

Enclosing a pattern within braces also allows the pattern to span lines. For example:

```
MATCH {[B,dd]
dd
[%,d]}
```

Identical to the example above, except that each pattern component is listed on a separate line.

Repetition

Repetition of any character or sets of characters can be indicated by an asterisk (*) or by a plus sign (+) following any pattern character or pattern group within braces. Plus (+) means that at least one occurrence of the pattern is required for the match; the asterisk (*) means that zero or more occurrences can be matched. These repetition indicators cannot follow items enclosed within brackets. Some examples:

`MATCH d+`	The plus sign indicates repetition of the digit, with at least one occurrence required for the match. Leaving the field blank is not acceptable unless the field type is 0 for

	optional, in which case a blank is accepted because processing specifications are ignored. For example, 2 or 7654321 or 55 are acceptable.
`MATCH Xd+`	This pattern accepts the letter X followed by one or more digits. For example, X1 or X2345, and so forth are acceptable, but not X.
`MATCH M{A,C,d}+`	A plus sign after a pair of braces indicates repetition of any item within the braces, in any order. Some acceptable values are MA, MCCC, or M12333CAA9.
`MATCH d*`	The asterisk indicates *optional* repetition that allows zero or more occurrences of the pattern. Thus, the digit can be omitted, or repeated any number of times. For example, nothing, or 3, or 123456, and so forth are acceptable matches.
`MATCH [d+]`	This pattern is another way of expressing the pattern shown above as d*.
`MATCH a*`	Accepts any alphabetic or empty string.
`MATCH Xu*`	This pattern accepts X alone, or followed by any number of uppercase letters. For example, XABC or XX or X are all acceptable.
`MATCH M{A,C,d}*`	Any of the enclosed characters can be repeated in any order, or can be omitted. Thus, M is acceptable, as are MAA, MCCAC12, MA63CCA5, and so forth.

Operator Hierarchy

Table 4-5. summarizes the operators allowed in a MATCH pattern. The pattern operators are evaluated in the following order, where x and y are any patterns:

```
Highest             !x          Transparency
   |                x:y         Range
   |                x+ or x*    Repetition
   v                xy          Concatenation
Lowest              x,y         Choice
```

Table 4-5. Pattern Match Operators

Operator	Function	Example
Exclamation point !	Transparency operator that allows use of any special MATCH characters as an element in the pattern.	`MATCH !u,!d,!,,` accepts any of the values `u, d,,, or !.`
Comma ,	Choice of subpatterns, any one of which satisfies the match.	`MATCH A,B, dd` accepts values such as `A,B, and 22.`
Colon:	Range of single characters in ascending collating order, any one of which satisfies the match.	`MATCH 2:6` accepts only the values `2,3,4,5, or 6.`
Braces {}	Required grouping that specifies at least one occurrence of any pattern within braces.	`MATCH {A,B}dd{%,d}` accepts `A223, B34%,A795`, and so forth.
Brackets [ ]	Optional grouping that allows zero or one occurrence of any item in pattern within brackets.	`MATCH [A,B]dd[%,d]` accepts `24,A99, 10%,123`, and so forth.
Plus +	Required repetition that specifies at least one or more occurrences of a preceding item, or a pattern within braces.	`MATCH Xd+` accepts values such as `X1,X22, X3334789`, and so forth, but not X. `MATCH {d,a} +` accepts values such as `11,A23,acb`, or `33ABC9`.
Asterisk *	Optional repetition that allows zero or more occurrences of a preceding item or a pattern within braces.	`MATCH Xd*` accepts values such as `X,X1, X22`, and `X3334789. MATCH {d,a}*` accepts a null value, or such values as `11,A23,acb`, or `33ABC9`.

Example

Some further examples of the MATCH statement are:

`MATCH 1dddd`	Accepts an integer between 10000 and 19999. Could also be expressed as: IN 10000:19999.
`MATCH [d]d!:dd [AM,PM]`	Accepts a time such as 3:00 PM or 12:00.
`MATCH {1:9,1(0:2}}!:{0:5}db*[{A,P}[M]]`	Accepts 12-hour clock time, such as 12:15 PM or 01:30 AM.
`MATCH {1:7}{0:7} *`	Accepts an octal number greater than zero with at least one digit and no leading zeros, such as 2047, or 1 or 24.
`MATCH ddd-dd-dddd`	Accepts any United States Social Security number, such as 044-24-0474.
`MATCH [(ddd)]b*ddd- dddd`	Accepts a United States phone number with an optional area code.

Processing Specifications For Advanced Processing

The advanced processing discussed in the rest of this section includes:

- **Data movement.** Data movement falls into two basic categories: setting the current field to a value and moving data between fields. Each of these categories uses the **SET** statement.
- **Data formatting.** When data is moved between fields with the SET statement, certain automatic formatting is performed. If the automatic formatting is not exactly what you want, or if you want to display data in a different format, you can use the FORMSPEC formatting statements to reformat the current data field during data collection. These statements are: STRIP, JUSTIFY, FILL, UPSHIFT.
- **Altering forms sequence.** The CHANGE statement allows you to alter the sequence in which forms are displayed.
- Conditional processing. The IF statement allows you to specify under which conditions the processing statements are to be executed.
- **Control Statements.** Two statements provide control over forms processing. The FAIL statement forces a failure of the current field edit, the STOP statement stops all processing of the current phase of the current form.
- **Processing phases.** Processing statements may be associated with one of four phases of form execution:

  — Configuration. Configure the terminal for specific field.

  — Form initialization. Determine initial values of fields.

  — Field edits. Edit and validate data entered in field.

  — Finish form. Complete processing of form.

For advanced processing, the sequence of specifications must be considered. If you need only the field edits described so far, you need not be overly concerned with the order in which statements are specified. Advanced processing statements, on the other hand, can be thought of as elements of a language where the order in which they are entered is important.

You may want to refer to "Phases" at the end of the section before reading the statement descriptions. In any case, you should be aware that all field edit and advanced processing statements can be executed in all phases, except configuration. Only configuration processing statements are executed in the configuration phase. Phases allow terminal configuration preceding initial processing for the form, which in turn precedes field edits and finish processing.

SET

Used to move data either by setting the current field to a value or by moving data between fields. Also performs automatic formatting.

Syntax

```
[SET {destination TO source
      destination
      TO source}]
```

Parameters

source	Can be a field name, save field name, constant, arithmetic expression, or an index retrieve operand within parenthesis. Refer to "Statement Syntax" earlier in this section for details.
destination	Can be a field name or save field name.

Discussion

When data is moved between constants, fields, and save fields, certain restrictions apply and certain conversions may take place. These depend entirely upon the data types of the source and destination. In general, any field, save field, or constant can be converted to a character type field, but numeric and date fields accept only data of a similar type. DIG fields accept only positive sources. If a source is $EMPTY, the destination is set to all blanks. Table 4-9. shows the conversion that is performed when data is moved between fields.

Assigning a Value to the Current Field

To set the current field to the value of another field, a save field, a constant, an arithmetic expression, or a value in a list located through an index, use the following SET statement:

Syntax

```
SET [fieldname] TO source
```

Parameters

fieldname	The current field name (default).
source	Can be a field name, save field name, constant, arithmetic expression, or an index retrieve operand within parenthesis. Refer to "Statement Syntax" earlier in this section for details.

Discussion

In general, any source value can be moved to a character type field. Numeric and date fields accept only data of similar type. If the field type is DIG, the source must be a positive value (see Table 4-7.).

By default, all fields are initialized to blanks in the initialization phase (refer to "Phases" at end of this section for more information on phases.) You can specify a particular initial value for any field by including an initial value in the Field Menu field attributes. More elaborate initialization can be done with this subset of the SET statement.

When you assign an initial value to a field in the Field Menu, you can specify only a constant. The constant is entered exactly like user input at a terminal. (Remember that input in FORMSPEC is NATIVE-3000; if a native language is specified, data is converted at run-time. Refer to Section 8.) When you assign a value through the SET statement, you have more flexibility. The values assigned are dynamic in that they may depend on values in other fields or in save fields, or they may be derived from an arithmetic expression, or through an indexed retrieval. If you do assign a constant through the SET statement, it must follow the rules for constants described earlier in this section. That is, a character string must be surrounded by quotes, a date string by exclamation points. Also, a date constant must be in the order MDY regardless of its destination format.

If a source field is of date type (MDY, YMD, DMY), the current field must be defined to be 8 bytes long to allow for internal delimiters. Some examples using the SET TO *source* statement follow.

Example

1. Assume a date field of the form MDY.

```
SET TO $TODAY                   \Sets the field to today's date in the
                                \format dd/dd/dd in the order of
                                \the field's date type.

SET TO !FEB 10,1986!            \Sets the field to the specified date.

SET TO DAT1                     \Sets the field to whatever value
                                \is in the field DAT1 at run-time.
```

2. Assume the current field is type DIG. The following statement sets this field to a digit selected from a list of digits by the index value, COUNT:

```
           SET TO COUNT OF 7,9,16,24,31,72,15,12
```

   If COUNT=5, the value assigned to the field is 31, if COUNT=3, the value 16 is assigned, and so forth.

3. Values may be passed from one form to another through save fields. Assume that when FORMA is executed, the save field SF3 is set to the value of F1. Further assume you are designing FORMB and want to set the current field to the value of the field F1 in FORMA.

```
SET TO SF3                      \Value passed from a field in a different
                                \form through the save field SF3.
```

Moving Data Between Fields

To move data to a field or a save field from another field or save field, or to move a constant, an arithmetic expression, or a value retrieved from a list to a particular field, use the following versions of the SET statement:

Syntax

[SET {*destination*
        destination "TO" *source*}]

Parameters

source	Can be a field name, save field name, constant, arithmetic expression, or an index retrieve operand within parenthesis. Refer to "Statement Syntax" earlier in this section for details.
destination	Can be a field name or save field name.

Discussion

When a source is not included, whatever value is in the current field is moved to the specified destination. (The current field is the field in which the SET statement appears.)

Example

1. Move the value resulting from an arithmetic expression to a numeric data field AMOUNT:

   ```
   SET AMOUNT TO 6 %(3*COST)
   ```

   This statement multiplies the value of the field COST by 3 and then sets AMOUNT to 6 percent of the result.

2. Set the save field SF3 to the current value of the field in which the SET statement appears:

   ```
   SET SF3
   ```

   Assume the current field is a character type with the value SMITH. The SET statement moves the value SMITH to the save field SF3. SF3 must be a character type save field.

Automatic Formatting

In general, automatic formatting performed during data movement is governed by the data type of the destination. The following discussion illustrates data movement for various data types. Refer to Table 4-9. for a summary of the conversion performed during data movement.

Character Type

If the destination is a character field, data moved to it does not change its relative position. If the source is too large for the destination, the data is truncated on the right when it is moved. If the source has fewer characters, the destination is padded with blanks on the right.

Table 4-6. Automatic Formatting for Character Data

Source	Destination
"ARMSTRONG" (9 characters)	"ARMSTRONGΔ" (10 characters)
"ΔΔARMSTRONG" (11 characters)	"ΔΔARMSTRON" (10 characters)
"ARMSTRONGΔΔΔΔΔ" (14 characters)	"ARMSTRONG" (9 characters)

Numeric Type

When data is moved between numeric fields, the following formatting is performed:

Sign	Any plus sign is stripped from the source before the number is moved to its destination. If the source is negative, a minus sign is inserted to the left of the first digit in the destination.
Decimal Point	If the source has an implied or actual decimal point (IMPn or NUM[n] data type), the fractional part is rounded and/or truncated or zero filled to conform to the number of decimal places specified for the destination. If the destination has no decimal position (NUMO, IMPO, or DIG), any fractional part is rounded and/or truncated. If the destination is NUM (floating decimal point), the number is right justified after being stripped of trailing zeros following the decimal point. If the source specifies no decimal places (type is NUMO, IMPO, or DIG) and if the destination has an implied or actual decimal point, the fractional part is zero filled. Note that if the length of the destination is too small, any decimal places are rounded and truncated until the source fits.
Commas	All commas in the source are removed.
Leading Zeros	Leading zeros are stripped in all cases.

The result is then placed, right justified, as shown in Table 4-8., in the destination field.

With Native Language Support, decimal and thousands indicators are language-dependent in the NUM[n] and IMP[n] fields. When data is moved between fields and automatic formatting occurs for data entered in any field, recognition, removal or insertion of these decimal and thousands indicators also depends upon the local custom for the native language specified for the forms file. The optional decimal symbols in constants will be local custom-dependent. For more information on Native Language Support, see Section 8.

Table 4-7. Automatic Formatting for Numeric Data

Source	Destination
123 (DIG)	123 (DIGIT,length is 5)
123 (IMP2)	1.23 (NUM2,length is 4)
12.3 (NUM1)	12.3 (NUM,length is 6)
12.3 (NUM1)	12.30 (NUM2)length is 5)
12.3 (NUM1)	12. (NUM2,length is 3)
12.3 (NUM1)	1230 (IMP2,length is 4)
12.3 (NUM1)	123 (IMP1,length is 3)
+3357 (NUM)	335700 (IMP2,length is 6)
-3357 (IMP3)	-3.4 (NUM1,length is 4)
001,000 (NUM)	1000.00 (NUM2,length is 7)

Date Type

Any date, regardless of the format of the source, is moved to the destination as `dd/dd/dd`, as shown in Table 4-8. Thus, if a date is going to be placed into a field using a `SET` command, the field must be at least eight characters long. The order depends on whether the destination is specified as `MDY`, `DMY`, or `YMD`. With Native Language Support, conversion from alphabetic months to the numeric destination month is language dependent. For more information on Native Language Support, see Section 8.

Table 4-8. Automatic Formatting for Dates

Source	Destination
FEB 5, 1986	02/05/86 (defined as MDY)
2/5/86	86/02/05 (defined as YMD)
2,5,86	02/05/86 (defined as MDY)
February 5, 1986	05/02/86 (defined as DMY)
September 16, 1986	09/16/86 (defined as MDY)
Oct. 23. 1986	86/10/23 (defined as YMD)
October 23, 1986	23/10/86 (defined as DMY)

Table 4-9. Conversion During Data Movement

From	**To**					
	CHAR	**NUM**	**NUMn**	**IMPn**	**DIG**	**DATE**
CHAR	Truncate or Pad with blanks on right	illegal	illegal	illegal	illegal	illegal
NUM	Truncate or Pad with blanks on right	Right justify; Strip leading zeroes; Pad leading blanks; try to fit 9 decimal places, rounding truncating, inserting a decimal point as needed	Right justify; Strip leading zeroes; Pad leading blanks; Round or truncate fractions, insert decimal point as needed	Right justify; Strip leading zeroes even if fractional; Pad leading blanks remove any decimal point.	Right justify; Strip leading zeroes; Pad leading blanks. Value must be positive)	illegal
NUMn						illegal
IMPn						illegal
DIG						illegal
DATE	Truncate or Pad with blanks on right	illegal	illegal	illegal	illegal	Convert to dd/dd/dd in order of destination; Left justify; Pad trailing blanks

Default Formatting

Besides the explicit formatting described below, data is formatted whenever it is moved between fields according to the rules for automatic formatting. If you want data entered by the user to be formatted automatically, you can specify the following version of the SET statement.

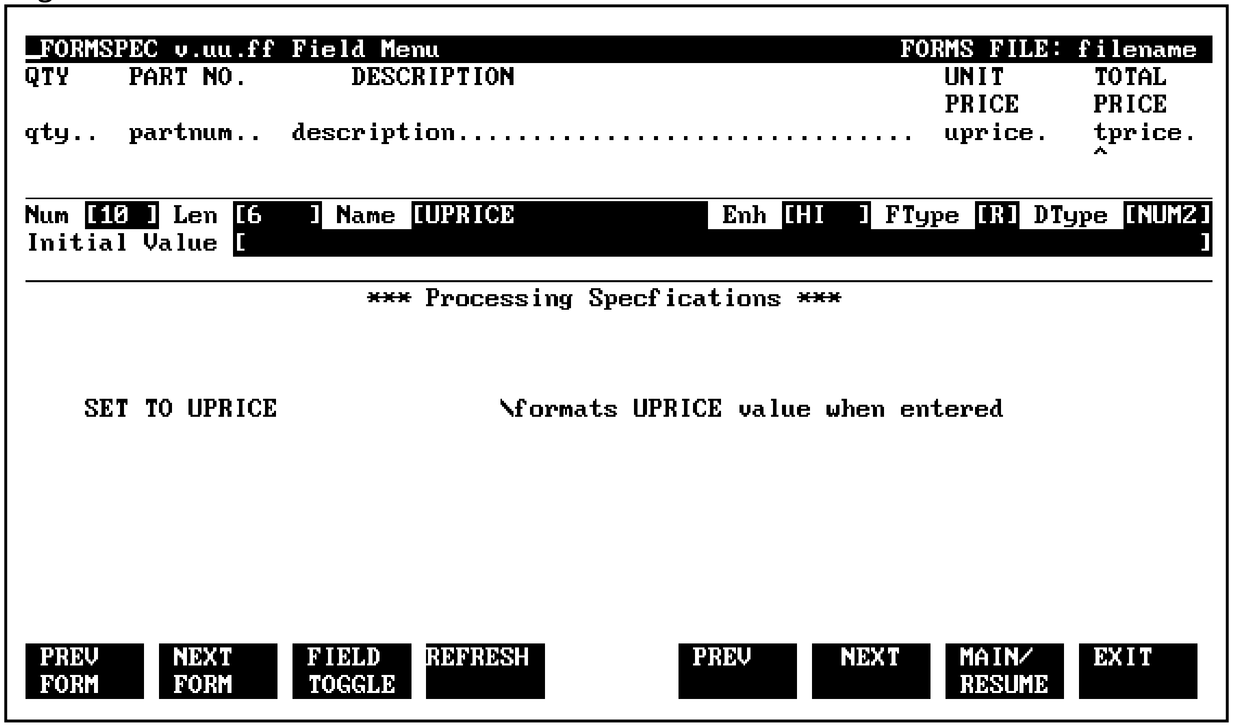
```
SET TO thisfield
```

Parameters

thisfield	is the name of the field in which SET appears. Essentially, you set the field to itself.

For example, you may want to ensure that a monetary value is always right justified with a decimal point inserted preceding two decimal positions. To do this, define the field as type NUM2 and then use the SET statement to force data entered in this field to be formatted. The Field Menu in Figure 4-3. illustrates this use of SET.

Figure 4-3. **Field Menu with a SET Statement**



```
_FORMSPEC v.uu.ff Field Menu                                FORMS FILE: filename
QTY    PART NO.       DESCRIPTION                              UNIT      TOTAL
                                                               PRICE     PRICE
qty..  partnum..  description.............................    uprice.   tprice.
                                                                         ^

Num [10 ] Len [6    ] Name [UPRICE          ] Enh [HI   ] FType [R] DType [NUM2]
 Initial Value [                                                               ]
------------------------------------------------------------------------------
                        *** Processing Specfications ***


    SET TO UPRICE              \formats UPRICE value when entered
```

Example

1. Suppose the user enters a price in the field as follows:

   `[123.5ΔΔΔΔ]`

   The SET statement causes it to be formatted as:

   `[ΔΔΔ123.50]`

2. In another situation, this version of SET TO can be used to format a date. Suppose the date field, DATE1, is type MDY and the user enters a date in the form:

   `[FEB 12, 86]`

   A SET statement of the form SET TO DATE1 in the Field Menu describing DATE1 formats the user entered data as:

   `[02/12/86ΔΔ]`

STRIP

Used to delete any specified character in the field.

Syntax

```
[STRIP {TRAILING
        LEADING
        ALL} "characters"]
```

Parameters

TRAILING	If TRAILING is specified, all occurrences of each character at the end of the field are replaced with blanks.
LEADING	STRIP LEADING replaces with blanks all occurrences of each character at the beginning of the field.
ALL	STRIP ALL deletes all occurrences of each character, compressing the data to the left.
characters	Any string of one or more characters can be specified. Since the statement does not apply to leading or trailing blanks, the statements STRIP LEADING or STRIP TRAILING are not useful.

Discussion

The STRIP statement can be used to format data entered into or moved to any field. (For automatic formatting, refer to the SET command earlier in this section.) As is true of all formatting statements, STRIP has an immediate effect on the data in the memory buffer; the formatted data is then available for display or copying into the application. When the user enters data into a field for which formatting is specified, the data is formatted as soon as **ENTER** or an appropriate function key is pressed. Thus, FORMSPEC formatting is done before data is written to the data or batch file. (In contrast, reformatting through REFORMAT, described in Section 5, is done after data is collected, edited, and written to the batch file.)

Note that only the characters between the leftmost nonblank and the rightmost nonblank characters are affected. That is, leading and trailing blanks are not included when the data is formatted. No formatting occurs when the field is blank since there are no characters to strip.

Example

Statement	Data Entered	After Formatting
STRIP ALL "-"	[548-72-2002]	[548722002ΔΔ]
STRIP LEADING "0"	[Δ000205000]	[ΔΔΔΔ205000]

JUSTIFY

Used to move data within a field to the left or right boundary of the field, or center it within the field.

Syntax

```
[JUSTIFY    {LEFT
             RIGHT
             CENTER}]
```

Parameters

LEFT	The data in the current field is shifted to the left.
RIGHT	The data in the current field is shifted to the right.
CENTER	The data in the current field is centered in the field.

Discussion

The JUSTIFY statement can be used to format data entered into or moved to any field. (For automatic formatting, refer to the SET command earlier in this section.) As is true of all formatting statements, JUSTIFY has an immediate effect on the data in the memory buffer; the formatted data is then available for display or copying into the application. When the user enters data into a field for which formatting is specified, the data is formatted as soon as **ENTER** or an appropriate function key is pressed. Thus, FORMSPEC formatting is done before data is written to the data or batch file. (In contrast, reformatting through REFORMAT, described in Section 5, is done after data is collected, edited, and written to the batch file.)

Example

Statement	Data Entered	After Formatting
JUSTIFY LEFT	[ΔΔSMITHΔΔΔΔ]	[SMITHΔΔΔΔΔΔ]
JUSTIFY RIGHT	[ΔΔSMITHΔΔΔΔ]	[ΔΔΔΔΔΔSMITH]
JUSTIFY CENTER	[ΔΔSMITHΔΔΔΔ]	[ΔΔΔSMITHΔΔΔ]

If the data cannot be centered exactly, the extra blank is on the right.

FILL

Used to replace all the blanks between the field boundaries and the first or last nonblank data character with a designated character.

Syntax

```
[FILL {TRAILING
       LEADING      } "character"]
```

Parameters

`TRAILING`	`FILL TRAILING` replaces those blanks following the data with the specified character.
`LEADING`	`FILL LEADING` replaces the blanks preceding the data with the specified character.
`character`	The specified *`character`* is any single character.

Discussion

The `FILL` statement can be used to format data entered into or moved to any field. (For automatic formatting, refer to the SET command earlier in this section.) As is true of all formatting statements, `FILL` has an immediate effect on the data in the memory buffer; the formatted data is then available for display or copying into the application. When the user enters data into a field for which formatting is specified, the data is formatted as soon as **ENTER** or an appropriate function key is pressed. Thus, FORMSPEC formatting is done before data is written to the data or batch file. (In contrast, reformatting through REFORMAT, described in Section 5, is done after data is collected, edited, and written to the batch file.)

If you want the field to remain blank, redefine the processing specification using a statement such as:

```
IF NE $EMPTY THEN FILL LEADING “0”.
```

Example

Statement	Data Entered	After Formatting
`FILL TRAILING` "*"	[Δ250ΔΔΔ]	[Δ250***]
`JUSTIFY RIGHT`	[Δ250ΔΔΔ]	[0000250]
`FILL LEADING` "O"		

Note that more than one formatting statement can be specified. Since the statements are executed in order of appearance, the `FILL` statement in the second example above affects the data justified by the preceding `JUSTIFY` statement.

UPSHIFT

Used to shift every lowercase character in a field to its uppercase equivalent.

Syntax

```
UPSHIFT
```

Discussion

The UPSHIFT statement can be used to format data entered into or moved to any field. (For automatic formatting, refer to the SET command earlier in this section.) This statement causes all alphabetic characters to be replaced with their uppercase counterparts. If edit statements expect data to be uppercase but it could be entered in lowercase, this statement should be executed prior to the edit statement. With Native Language Support, if a native language ID has been specified in the forms file, the UPSHIFT formatting statement will use native language upshift tables. For more information on Native Language Support, see Section 8.

As is true of all formatting statements, UPSHIFT has an immediate effect on the data in the memory buffer; the formatted data is then available for display or copying into the application. When the user enters data into a field for which formatting is specified, the data is formatted as soon as **ENTER** or an appropriate function key is pressed. Thus, FORMSPEC formatting is done before data is written to the data or batch file. (In contrast, reformatting through REFORMAT, described in Section 5, is done after data is collected, edited, and written to the batch file.)

Example

Suppose the field contains an United States state code that is to be checked against a list of uppercase state codes. The following statements ensure that codes entered in lowercase pass the edit:

```
UPSHIFT
IN "NY","NJ","PA"
```

CHANGE

Used to change the specification of either the current form or the next form.

Syntax

```
[CHANGE NFORM TO {[CLEAR
                   APPEND
                   FREEZE APPEND] ["formname"
                                   fieldname
                                   indexretrieve
                                   $RETURN
                                   $HEAD$END
                                   $REFRESH        ]}
 CHANGE CFORM TO {NOREPEAT
                  REPEAT
                  REPEAT APPEND}]
```

Parameters

NFORM	indicates the next form to be displayed after the current form at execution time.
CFORM	indicates the current form.
CLEAR	indicates that the screen is to be cleared when the next form is displayed.
APPEND	indicates that the next form is to be appended to the current form.
FREEZE APPEND	indicates that the current form is kept on the screen when the next form is appended to it even after the screen is full, at which point the top next form is rolled off the screen.
NOREPEAT	indicates that the current repeating form is to be stopped.
REPEAT	indicates that the current form is to be repeated.
REPEAT APPEND	indicates that the current form is to be repeated and appended to itself.
formname	is the name of any existing form in the forms file.
fieldname	is the name of a character type field that contains a form name.
indexretrieve	is an item in a list of existing form names. It is specified as: *index* OF *"name" [,"name"]...* where *index* is an integer, and *name* is a form name or any

	field or save field whose value is a form name. The *name* must identify an existing form.
`$RETURN`	indicates the last different form displayed before the current form at execution time.
`$HEAD`	indicates the first form displayed at execution time.
`$END`	indicates to ENTRY or an application that the current form is the last form to be displayed.
`$REFRESH`	indicates that current form is to be refreshed (cleared of entered data) and displayed as the next form.

Discussion

This statement may be entered in a processing specification for any field in the form. It causes the specified changes to the current or next form to take effect when the next form is requested, and it causes the specified next form to be displayed when the current form is finished at execution time.

If several NFORM statements are specified in a form, only the last statement executed is effective.

When forms sequence is defined in the Form Menu (refer to Section 3), the current form may be repeated, or repeated and appended to itself, or neither, when the next form is requested. Also, the current form may be cleared when the next form is requested, or it may remain on the screen with the next form appended to it. The CHANGE statement allows you to alter these form specifications dynamically.

Additionally, the CHANGE statement allows you to specify a different next form than the one specified on the Form Menu. Note that the form changes do not occur when the field is entered with the ENTER statement, but only after the current form has been finished.

Example

```
CHANGE NFORM TO CNT OF "FORM3", "FORM4", "FORM5", "FORM6""FORM7"
```

Depending on the current value of field CNT, the next form displayed is one of the forms in the list. For instance, if CNT is 3, FORM5 is the next form.

```
CHANGE NFORM TO $END
```

After this form is finished, no more forms will be displayed.

```
CHANGE NFORM TO APEND "FB"
```

The next form, FB, is to be appended to the current form.

IF

Used to execute any of the processing statements only under certain conditions.

Syntax

```
[IF condition THEN [statement]
    [statement]
        .
        .
         .
      [statement]

[ELSE [statement]
       [statement
             ].
              .
                .
         [statement]]]
```

where *condition* is:

```
[{ constant
   field
   save field
   expression
   (indexretrieve) }
                           editstatement]
```

Parameters

condition	A condition specified in an `IF` statement can be any edit statement described below. The edit statement can be preceded by a constant, a field name, a save field name, an expression, or an index retrieve operand within parentheses. Refer to "Statement Syntax" earlier in this section for details. If an operand precedes the edit statement, that value is tested; otherwise, the value of the current field is tested. If editing statement passes, then the condition is true. If it does not, then the condition is false. Note that this differs from interpretation of a standalone edit statement, which causes the field to return an error and stops field processing if the data fails the edit. Only edit statements can be used in conditions.
editstatement	A condition specified in an `IF` statement can be any edit statement `MATCH` and `CDIGIT`
statement	Any of the processing statements can be executed conditionally depending an the run-time interpretation of the specified condition. Refer to the syntax rules below.

Discussion

An IF statement consists of two groups of statements: the THEN part and the ELSE part. Either may have no statements associated with it. The THEN part may include statements on the same line as THEN, plus statements indented from the IF on immediately following lines. An ELSE statement at the same level of indentation as the IF corresponds to that IF. Like the THEN part, the ELSE part can have statements on the same line, plus statements on immediately following lines that are indented from it. Nested IF statements must be indented from each enclosing IF, but otherwise follow the same rules. Table 4-10. illustrates some variations on the IF statement according to the syntax rules, which are:

- No more than one IF or ELSE statement may appear on the same line.
- When non-IF statements follow either the THEN or the ELSE, they may be on the same line as the THEN or ELSE. Statements can be separated from each other by an optional semicolon (;).
- The entire ELSE portion of the statement may be omitted. In such a case, no statement is executed if the condition is false. If ELSE is included, it must be the first statement following the THEN part that is at the same level of indentation as its corresponding IF.
- Nested IF statements are allowed to a maximum of eight levels of nesting. They must maintain nested indenting.
- When nested IF statements are specified, they must be indented. The indenting is essential for multiple statements to identify the scope of the THEN part, as well as the ELSE part.
- An IF statement (including the THEN and ELSE part) must not cross phase boundaries. (Refer to "Phases" later in this section.)

Example

`IF QUANTITY GT 100 THEN…`	If the current value of the field QUANTITY is greater than 100, any statements in the THEN part at the same level are executed.
`IF NE $EMPTY THEN…`	If there is a value in the current field (it is not blank), then any statements in the THEN part are executed.
`IF SAV1 IN 12:50,100:120 THEN…`	If the value of the save field is within the range 12 through 50 or 100 through 120, then any statements associated with THEN are executed.
`IF MINLEN 1 THEN…`	If at least one character was entered in the current field, execute any statements associated with THEN.

Table 4-10. Variations on the IF Statement

Variation	Description
IF *condition* THEN *statement*	Simple IF statement. If *condition* is true, then *statement* is executed; if false, then *statement* is not executed.

IF *condition* THEN
    statement
    statement
    statement
	If *condition* is true, all three *statements* are executed in the order specified. If *condition* is false, none of the three *statements* is executed.

IF *condition* THEN *statementA*
ELSE *statementB*
	If *condition* is true, *statementA* is executed; otherwise, *statementB* is executed.

IF *condition* THEN *statementA*
    statementB statementC
ELSE *statementD*
	The *statementA, statementB* and *statementC* are executed if the *condition* is true; *statementD* is executed if the *condition* is false,

IF *condition* THEN *statementA*
    statementB
    statementC
ELSE *statementD*
	If condition is true, *statementA, statementB*, and *statementC* are executed; otherwise *statementD* is executed.

IF *condition1* THEN
   IF *condition2* THEN
     statementA
     statementB
ELSE
    IF *condition3*
     statementC
     statementD
ELSE
     statementE
	Only if *condition1, condition2* are both true, *statementA, statementB* are executed.
	Only if *condition1, condition3* are true, but *condition2* is false, is *statementC* executed.
	Only if *condition1* is true, regardless of whether *condition2, condition3* are true, is *statementD* executed.
	Only if *condition1* is false, is *statementE* executed.

FAIL

Used to force failure of a field edit.

Syntax

[FAIL ["*message*"]]

Parameters

message	Specifies a custom error message.

Discussion

When this statement is executed, it forces data entered in the current field to fail any specified edits. When FAIL is executed, an error flag is set for the field. If you include the message parameter, that message is issued; otherwise, a system message is issued.

A FAIL statement is normally used in an IF statement where it is executed conditionally.

Example

For example, suppose you want to ensure that an entered value is in a table of values. You can use the FAIL statement to ensure that no further edits are performed and an error message issued if the value is no found.

```
IF NIN $STATE THEN
   SET TO $EMPTY
   FAIL "Must enter legitimate state code"
```

If the value is found in the table, the FAIL statement is not performed nor is the field cleared to blank.

STOP

Used to stop processing the current phase of the current form.

Syntax

[STOP]

Discussion

When STOP is executed, no further processing is performed on the current field, or on any subsequent fields in the form in this phase.

Example

For example, if using ENTRY, processing of the entire forms file could be terminated when "END" is entered in the current field with the following specification:

```
IF EQ "END" THEN
   CHANGE NFORM TO $END;STOP   These statements terminate execution
                               of the forms file in "END" is the
                               value of the current field.
```

The STOP command ends processing of this form; using the $END constant as the next form instructs ENTRY to end processing of all forms.

PHASES

Used to determine when specified processing specifications are executed.

Syntax

```
phaseheader
[statement]
    .
    .
    .
[statement]
```

Parameters

phaseheader One of the following headings. The phase headings must be specified in the order shown:

CONFIG	Perform the following configuration during form presentation.
INIT	Perform the following statements during initialization (VINITFORM).
FIELD	Perform the following statements during the field edit phase (VFIELDEDITS).
FINISH	Perform the following statements during the finish phase (VFINISHFORM).

Configuration processing specifications must always be preceded by the phase header CONFIG. The INIT header can be omitted if the initialization processing specifications are followed by the FIELD phase header and field edit processing specifications. The FIELD phase header can be omitted if there are no configuration or initialization processing specifications. The FINISH header must always be specified to indicate that a statement is to be executed in that phase.

statement A processing statement to be executed in the specified phase.

Discussion

During the configuration phase, local edits are loaded to terminals that support the local edit feature (see Appendix G), and the output device and error lights to be lit are identified for data capture devices. During the initialization phase, all fields in the form are initialized; during the field edit phase, all fields (in screen order left to right, top to bottom) are validated and edited. An application usually requests execution of the finish phase only when all fields have passed the edit tests of the field phase. Finish statements apply to the entire form. (Refer to Figure 4-1.for an illustration of the flow of form execution under ENTRY control.)

You can specify that a statement be executed during a particular phase with the phase specification statements. It is important to note that while specifications are entered field by field, execution of the phases applies to the entire form. If you omit these phase headings altogether, all statements are executed in the field edit phase.

Example

```
CONFIG
LOCALEDITS alphabetic
LIGHTB
DEVICE printer
INIT
SET TO 20.00
FIELD
IN 20:200
IF IN 100:200 THEN JUSTIFY RIGHT
FINISH
SET TO SAVE_FIELD_1
```

If you want statements to be executed in a particular phase, you must precede the statements by the appropriate phase headings. Phase headings must be specified in the order shown above.

Configuration Phase

If you use the configuration phase, the `CONFIG` statement must be the first statement in the processing specification area on the field menu. The `CONFIG` statement is processed only for the field in which it appears. The commands that can be used in the configuration phase are: `DEVICE, LIGHT` and `LOCALEDITS`. (Refer to the "Processing Specifications for Configuration" earlier in this section.)

Initialization Phase

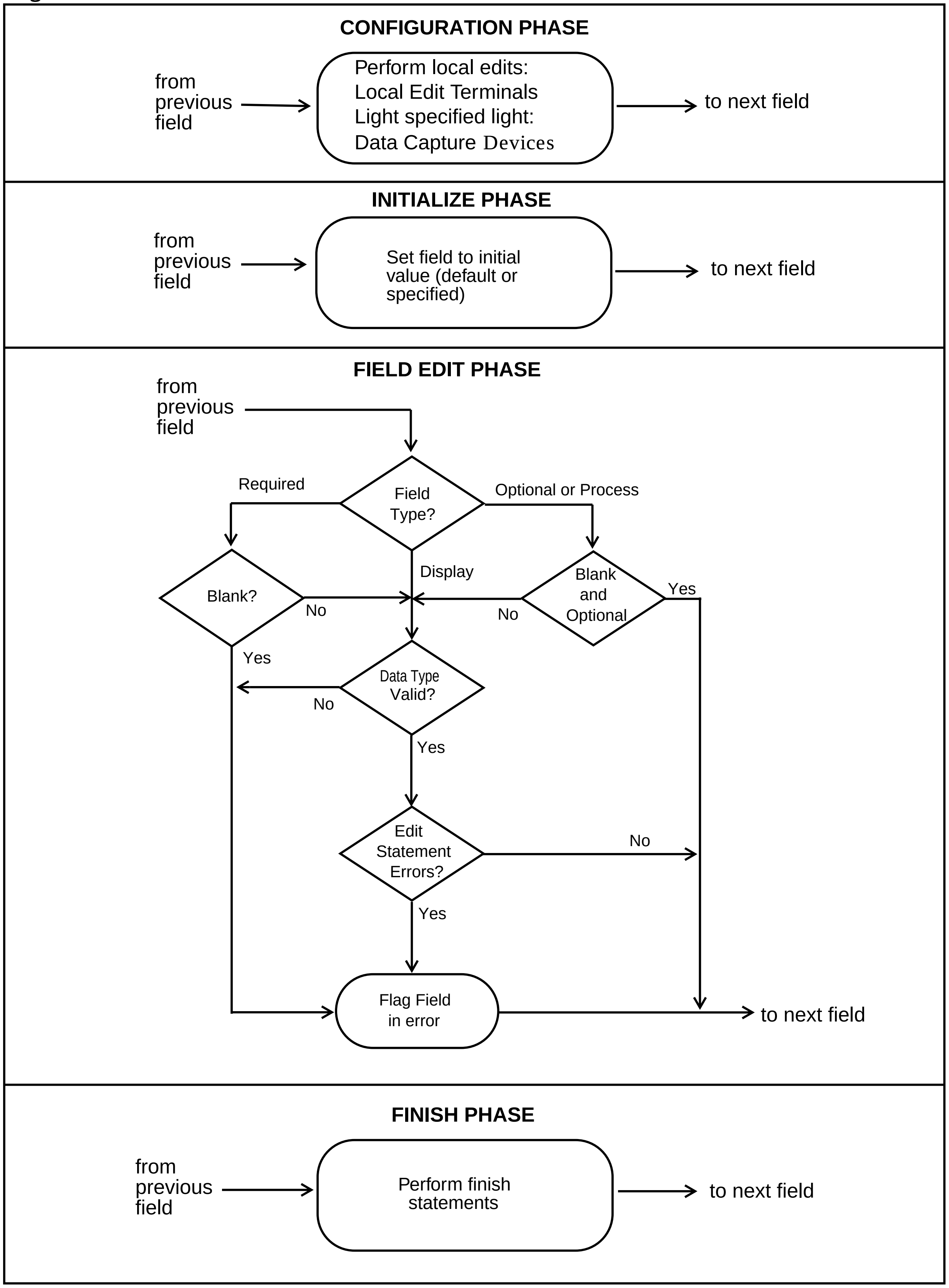
The initialization phase occurs when `VINITFORM` is called. The processing that precedes the display of the form is usually performed in this phase.

For each field in the form, the field is first initialized to any value specified in the Initial Values box of the Field Menu. (By default, all fields are set to `$EMPTY`, all blanks). If any initialization statements are included in the field processing specifications, these are executed next. This process continues until all fields in the form are initialized.

If the form being initialized is a child or sibling to the previous form, data from the previous form is transferred to this form (with conversion if necessary) before initializations occur. Refer to "Form Families" in Section 3 for more information.

A `SET TO` source statement may be used to initialize field values.

Figure 4-4. Flow a Phase Execution for ENTRY



Field Edit Phase

The field edit phase occurs when VFIELDEDITS is called. The field edit phase is usually requested by the application after the form is displayed on the screen, and the user has input data. During this phase, the entered data is checked according to the field attributes and any processing specifications entered by the designer. The Edit statements can be executed only in the field edit phase.

Each field in turn is examined. The first action is to check the field type. If it is optional and the field is blank, the rest of the field edit phase for this field is skipped. If the field is required, it must contain a nonblank value. If it has a value, the data is checked to see if it conforms to the field's data type. If it does, then any field processing statements are executed in the order they were entered.

Edit failures detected in this phase cause the field to be flagged in error so the user can be informed. All further processing on this field is stopped when an error is detected, and control passes to the next field.

ENTRY loops through all processing statements in the field edit phase until all errors have been corrected by the user.

Finish Phase

The finish phase occurs when VFINISHFORM is called. The finish phase usually is requested by the application after all data has been entered and validated for the entire form. The ENTRY program executes statements in this phase only when no errors are detected after **ENTER** is pressed. Finish statements direct irreversible processing during which the global environment can be altered.

Under ENTRY control, any processing performed by the preceding two phases can be undone prior to the FINISH phase. For instance, if the user presses REFRESH, the initial values replace any that were entered. In the finish phase, such changes can no longer be made.

Under ENTRY control, the batch file is written as soon as the finish phase is complete. The assumption is that all fields have been tested and corrected; fields with errors are not enhanced in the finish phase.

5 Reformatting Specifications

Information gathered during data collection is stored in one or more "batch" files. The data entered for each field in a form is stored exactly as it was entered. All blanks, punctuation, or special characters are included in the data. The only exception is that any formatting specified on the Field Menu or any automatic formatting resulting from data movement (refer to Chapter 4) is completed before the data is written to the batch file. Thus, if a value has been entered in the middle of a field, the blanks on either side are stored as part of the data; or if a number has been right justified and zero filled, it is stored in the batch file in that form.

All data from all fields in a single form are concatenated together without delimiters into a single record in the batch file.

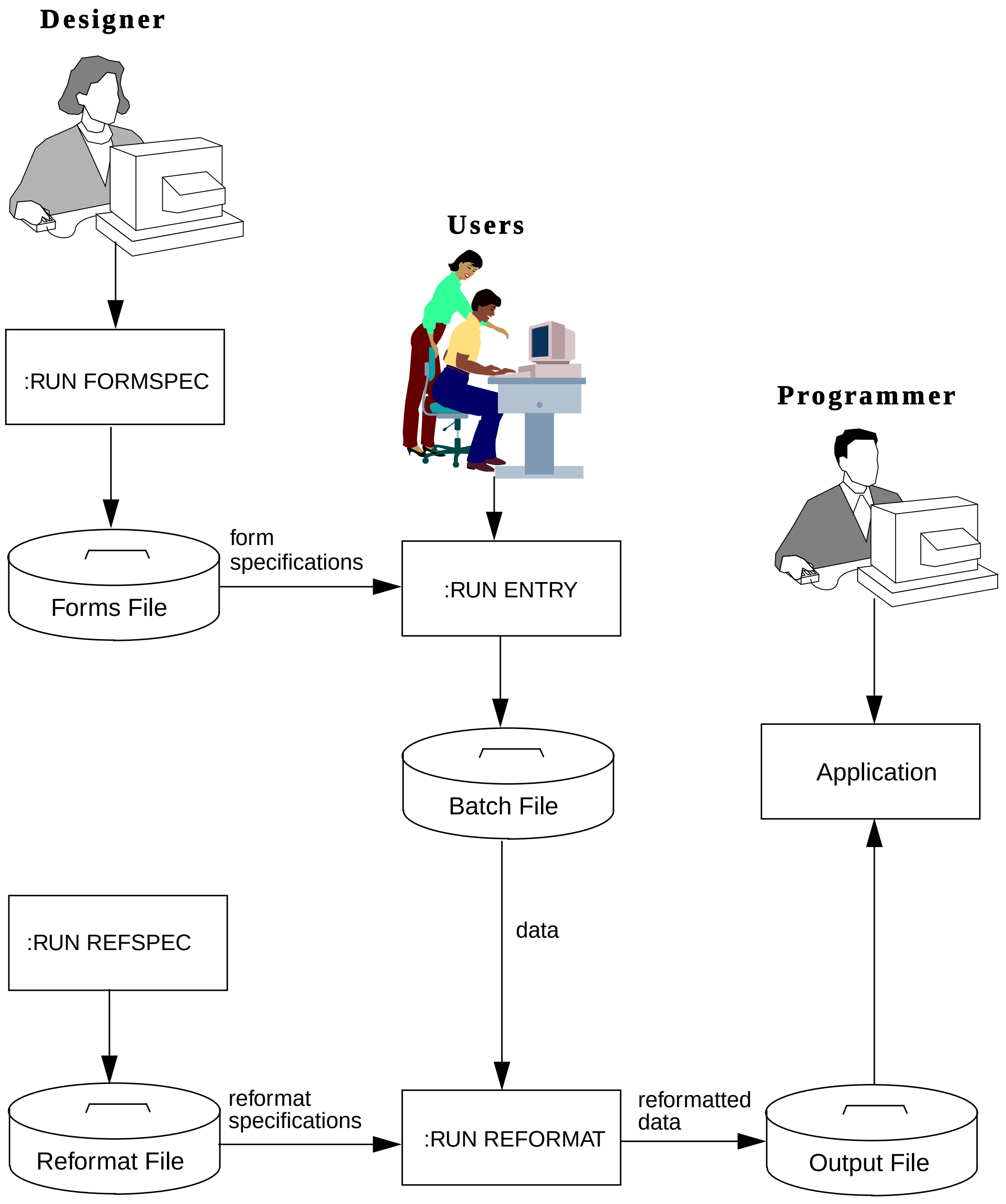
If the configuration of the data in a batch file is not suitable for input to an application program, the data can be reformatted with REFORMAT. This program writes the data from the batch file to another file, the "output" file, according to formatting specified through the reformat design program, REFSPEC. (Refer to Figure 5-1.)

You may want to reformat data entered in the batch file in order to:

- Combine data from several forms into one record in the output file.
- Separate data from a single form into two or more output records.
- Generate several output files from the data entered through a single forms file.
- Add literals (such as sort codes or record separators) to each output record.
- Use only a selected portion of the data entered in any field.
- Write only selected fields from any form to an output record.
- Fill, justify, or strip characters from data entered in any field.
- Add a check digit to alphabetic or numeric data.

This chapter describes how to use the reformat design program REFSPEC. It is an extension of forms design and is used by the forms designer in conjunction with an application programmer to specify how data is to be reformatted. Use of REFORMAT is described at the end of this chapter.

Figure 5-1. Relation among Files Used for Formatting



Files

One batch file containing collected data can be reformatted into one output file. This output file is then used as input to the application that processes the collected data. Data from more than one batch file can be written to one output file in sequence — that is, data from a batch file can be appended to data from another batch file in an existing output file with an MPE `:FILE` command equation. (See "Concatenating Batch Files", this chapter, for an example of this.) Without a file equation, data from each batch file overwrites data in an existing output file.

The reformatting specifications themselves are stored in a "reformat" file. The specifications in this file determine how the data in the batch file is to be stored in the output file.

One output file is generated for each reformat file. If you wish to separate the data into more than one output file, you must establish different reformat files, and run the reformatter program separately, to generate each output file. REFSPEC accepts either KSAM or standard MPE reformat files. The only forms files created by REFSPEC, however, are MPE files. In the MPE reformat files, the key and form records are interspersed throughout the file. MPE reformat files do not require extra data segments and are automatically recovered after a power failure or system crash. Refer to Appendix H for information on converting KSAM files to MPE files.

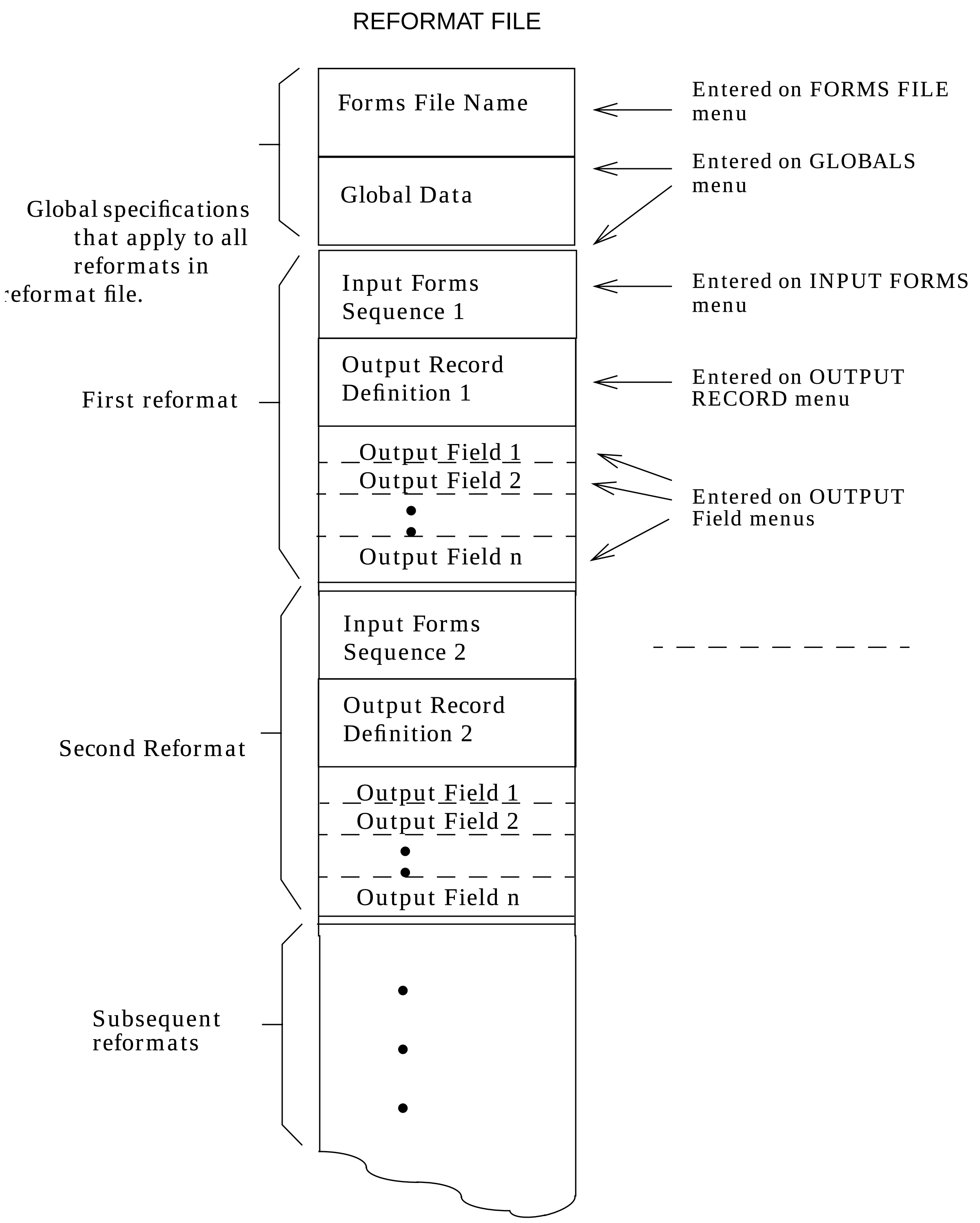
Reformat File

The reformat file consists of the following components:

Forms File Name	Identifies the forms file that contains the forms through which data in the batch file was entered.
Global Information	Specifications that apply to the entire output file, such as a string used for an end-of-record mark, or record length and whet her the record length is fixed or variable. Any or all global specifications can be omitted since defaults are provided.
Input Forms Sequence	Specify the form names on which data to be reformatted was entered. Only data from the forms in an input forms sequence is written to the output file. At least one input forms sequence must be included, and as many may be included as there are forms in the forms file.
Output Record Definition	Specify each field to be written to the output file, and define the beginning of each output record. Usually, one output record definition follows each input forms sequence.
Field Specifications	Special reformatting at the field level can be specified for each field listed in the preceding output record definition.

This information is entered on menus very much like those issued for FORMSPEC. Figure 5-2. illustrates a prototype REFORMAT file.

Figure 5-2. Prototype of REFORMAT File



Relation of Forms to Output Records

Every record in a batch file consists of data entered on a single form. It may be that the data on one form represents a logical group of information. However, the data from a sequence of forms may make up such a logical grouping, or the data entered on one form may make several logical groupings. The reformatter allows you to rearrange the entered data into output records with different groupings.

Before running REFSPEC to set up your reformat file, it is important to understand the relations between the forms (and the data entered on the forms) and the output records generated by the reformatter. The forms to be reformatted are identified in the input forms sequence of the reformat file; the output records are defined in the associated output record definitions.

Input Forms Sequence

Each input forms sequence lists the form or forms on which the data to be reformatted was entered.

In order to generate the output file from the reformatting specifications, REFORMAT reads each batch record in turn. Associated with each record is the form on which the data was entered. For the first batch record, REFORMAT searches the input forms sequences in the reformat file for a matching "reformat identifier". This identifier is the name of the first form in each in put forms sequence, and must be unique in the reformat file. If a matching reformat identifier is not found, the batch record is skipped.

Other form names may follow the reformat identifier in the same input forms sequence. These form names need not be unique. If a sequence of forms is specified, the batch records following the record that matches the reformat identifier must exactly match the form names in the input forms sequence. If they do not, the batch record is skipped, and the next batch record is checked against all reformat identifiers in the reformat file. Some rules to remember:

- Every batch file record contains data entered on a single form.
- The form name on which the data was entered is stored in the batch record with the data.
- Records in the batch file are processed sequentially starting with the first record and continuing through to the end.
- The first form name in each input forms sequence is the reformat identifier, and must be unique.
- In order to be reformatted, the form name of a batch record must match one of the reformat identifiers or be in a sequence following a form that matches a reform at identifier.
- Form names in an input forms sequence following the reformat identifier can appear in other input forms sequences as reformat identifiers or as part of a sequence following the reformat identifier.
- If more than one form is named in an input forms sequence, the sequence of batch

records starting with the reformat identifier must exactly match the sequence of forms.

Output Record Definition

The output record definition determines how information from batch file records is stored in the output file. Any field in the batch file that is to be written to an output record must be uniquely identified. The sequence of the output field names in the output record definition determines the order in which fields are written to the output record. Fields from the batch file may thus be reordered, or repeated, or omitted from the output record.

Constants, in the form of literals, system constants, or the numeric equivalent of USASCII characters, can be interspersed freely between fields or portions of fields in the output record definition.

The output record definition also determines where each output record begins. The designer can mark one or more fields (or constants) as the starting point of a record. If no field is so marked in an output record definition, then the first field in the definition is appended to the last field in the previous definition as part of the same record. Thus a batch record can be divided into several output records, or be made part of a larger record.

Note that you can specify a fixed record length, or a maximum variable record length, for all output records through the Globals Menu. REFSPEC does not allow the designer to specify a record in any one output record definition that is longer than this maximum length; however, it does not check the length of a record formed from several output record definitions. Some rules to remember:

- A unique output field name must be assigned to any batch field to be written to the output file. (Usually, the input field name is unique and can be used as the output field name.)
- Batch fields are written to the output file in order of the output field names.
- Only those fields specified in an output record definition are written to the output file.
- A particular batch field can be specified many times in a single output record definition as long as each occurrence is given a unique output field name.
- A start-of-record marker in an output record definition determines the start of an output record.
- A start-of-record marker can be associated with any field or constant in an output record definition.
- If no start-of-record marker is included in an output record definition, the first field of this definition follows the last field of the preceding definition when it is written to the output file.

Field Specifications

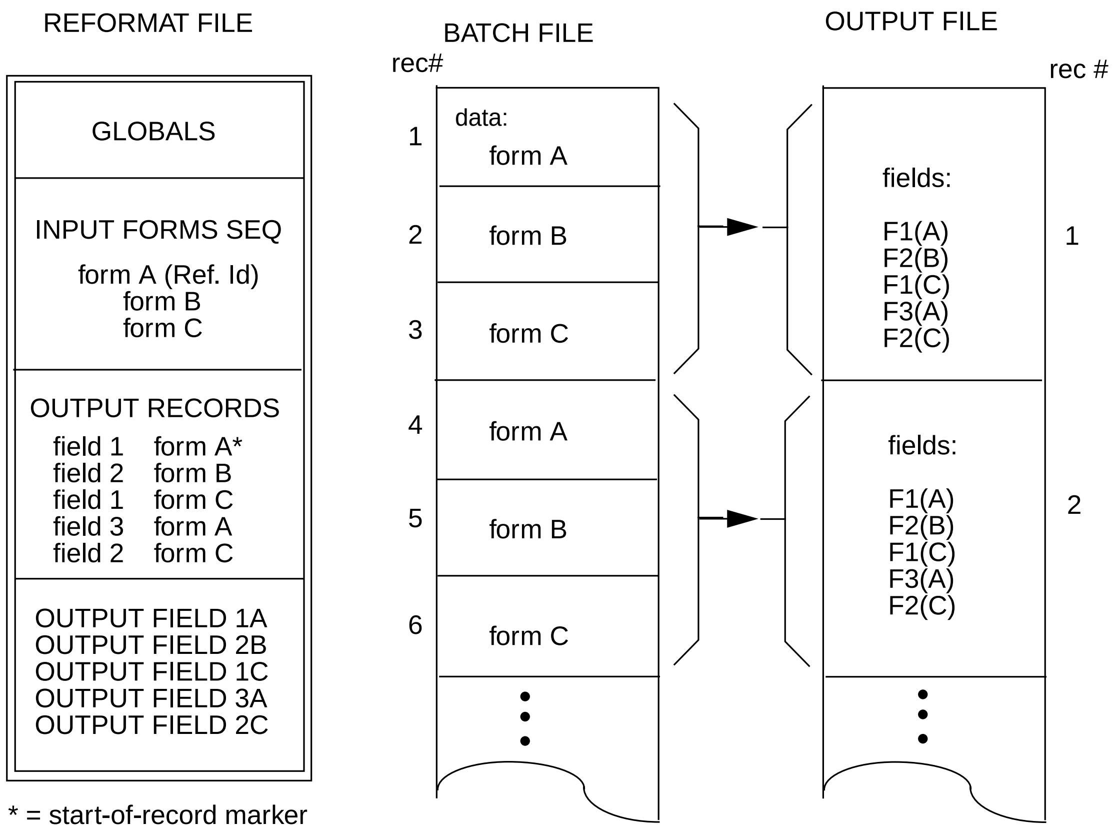
An Output Field Menu is issued for each field named in the preceding output record definition. The designer can reformat the output field, or leave the field as it was written to the batch file.

The following set of examples illustrates some ways in which the data from a batch file can be reorganized in an output file.

Combining Data from Several Forms

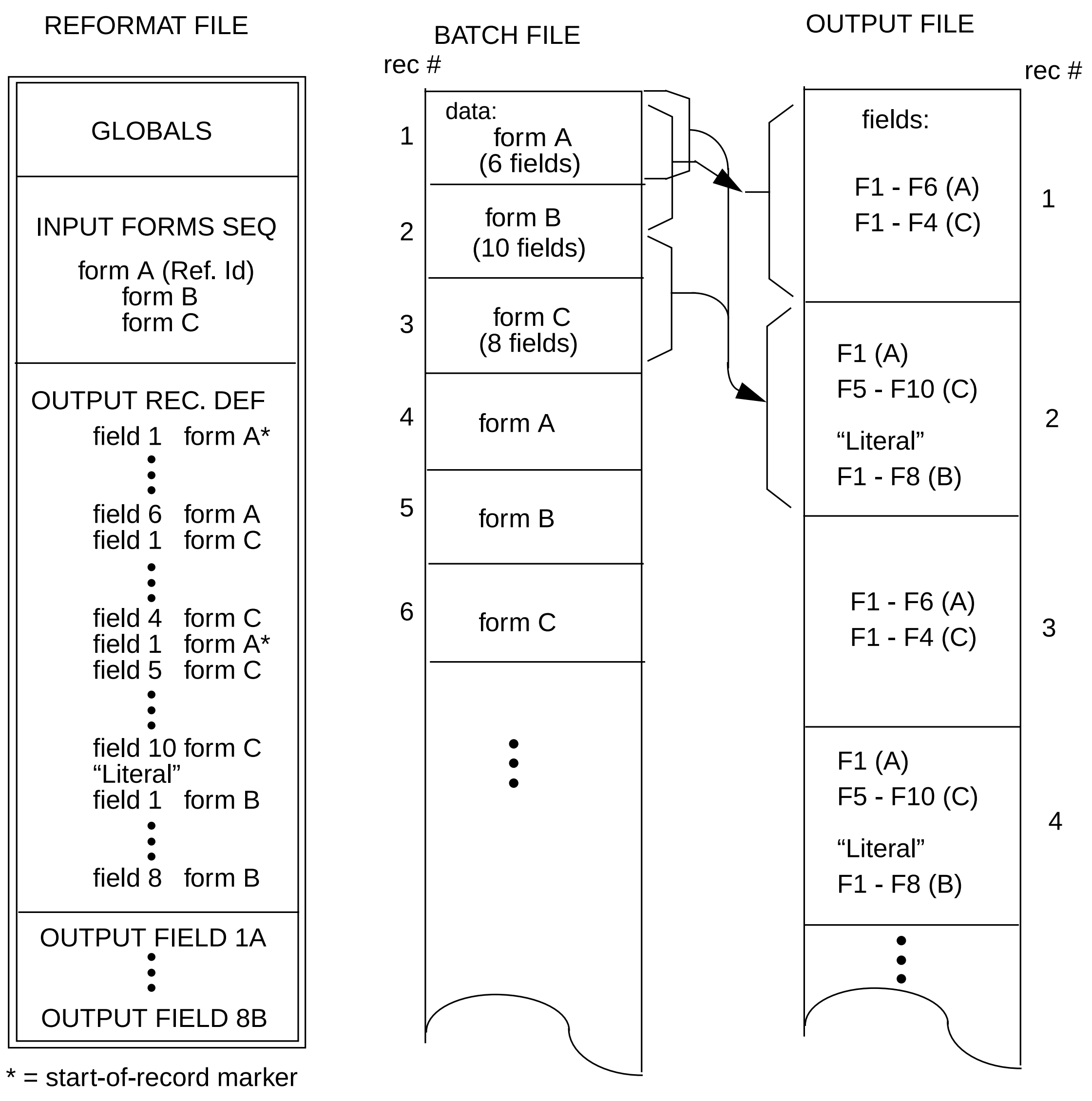
Suppose data is entered on forms A, B, and C. The reformat file shown in Figure 5-3. processes these records in sequence and writes them to one output record.

Figure 5-3. Combining Data from 3 Forms into 1 Output Record



Data from the batch file illustrated in Figure 5-3. can be reformatted in many different ways. For example, Figure 5-4. shows how this same data could be combined into two output records with data from forms A and B appearing in both records. Note that when the same field appears more than once, it must be given a unique field name for each occurrence (this is not specifically shown in Figure 5-4.)

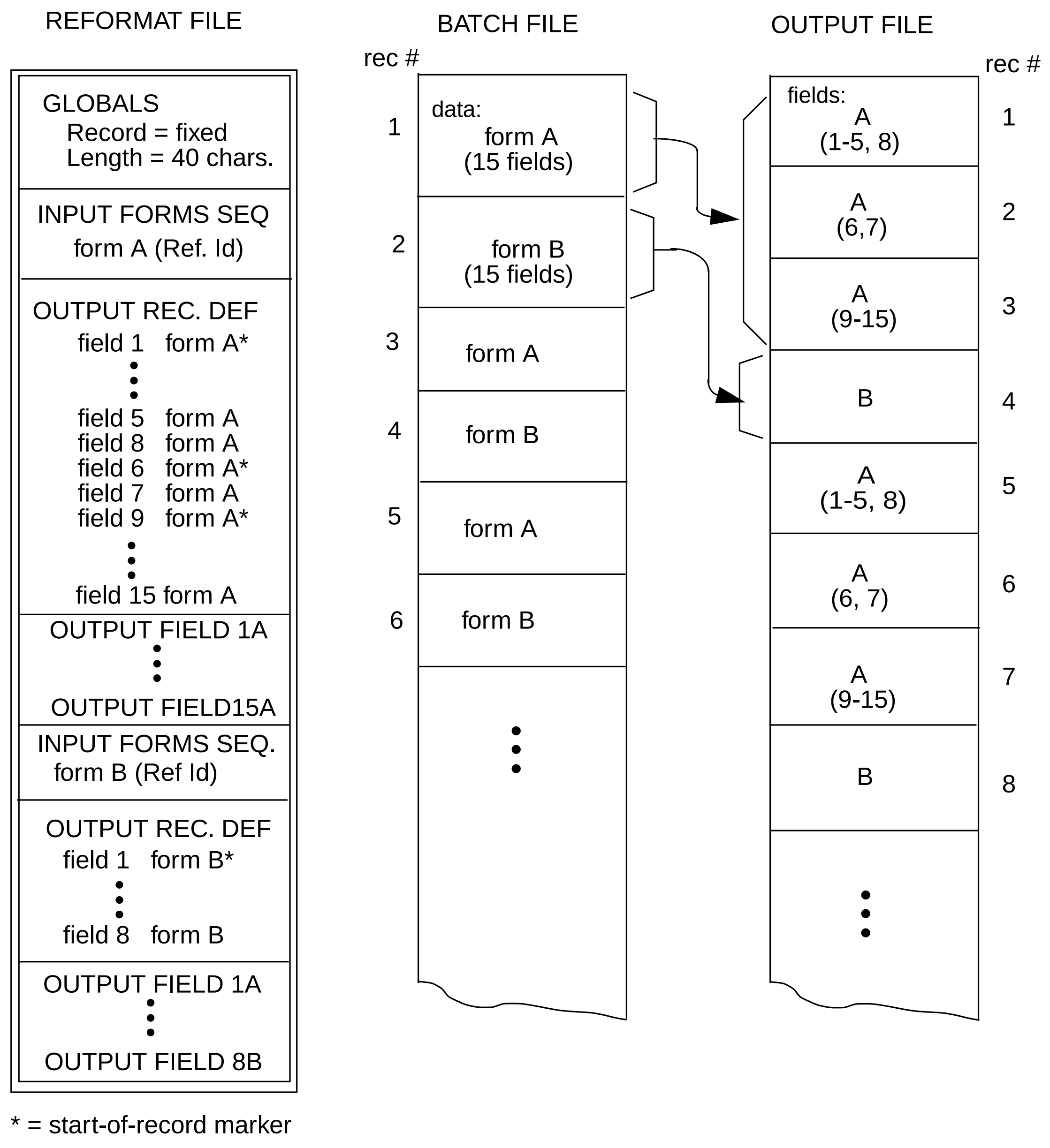
Figure 5-4. Reformat Data from 3 Forms into 2 Output Records



Separating Data into Several Records

In the following example, Figure 5-5., data entered on form A is separated into three output records, each a fixed length of 40 bytes. Data from a subsequent form is written to a single 40-byte record. If any group of data is less than the 40-byte record length, the record is padded with blanks. The data cannot be formatted to exceed the record length.

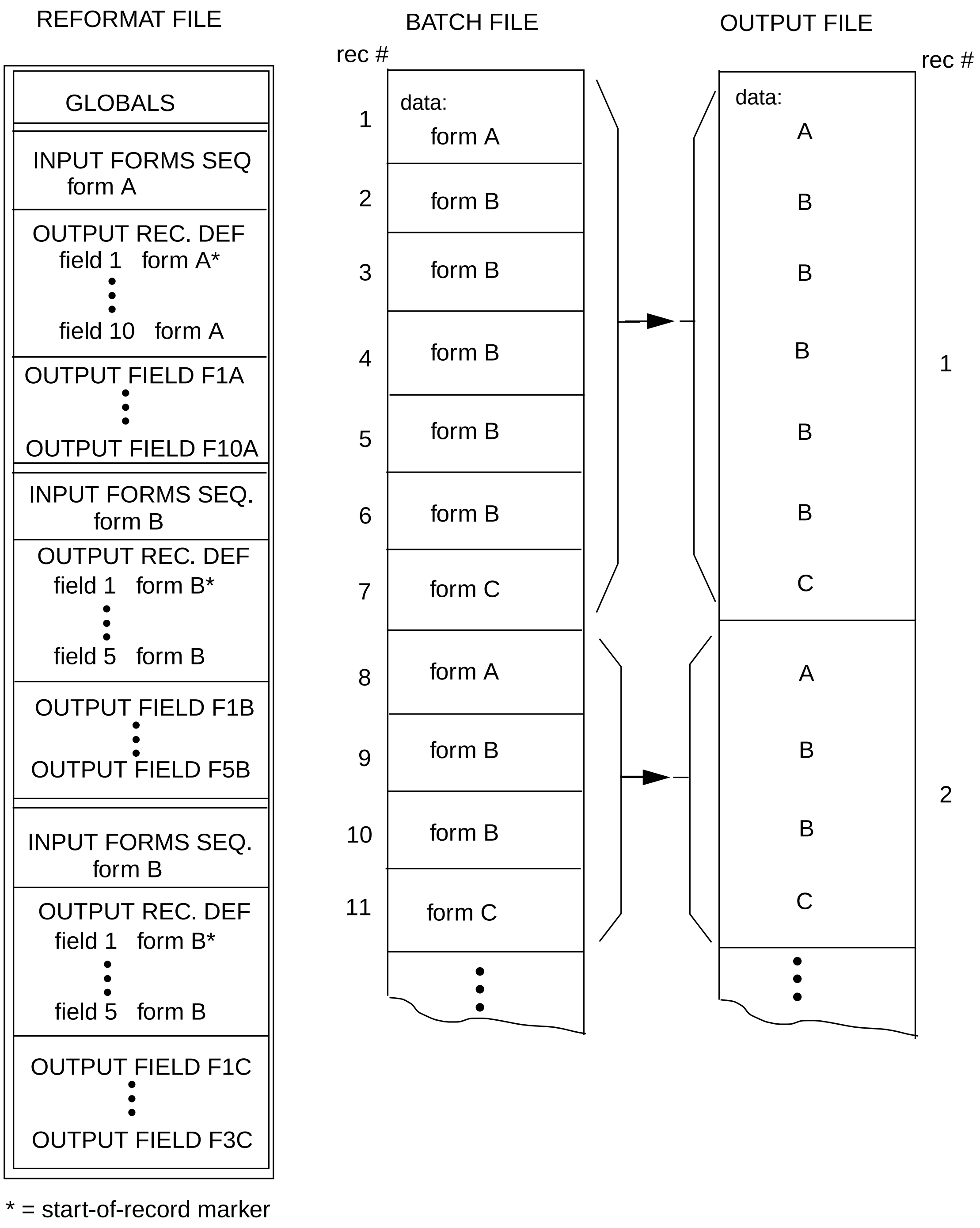
Figure 5-5. Separating Data from 1 Form into Several Records



Reformatting Repeating Forms

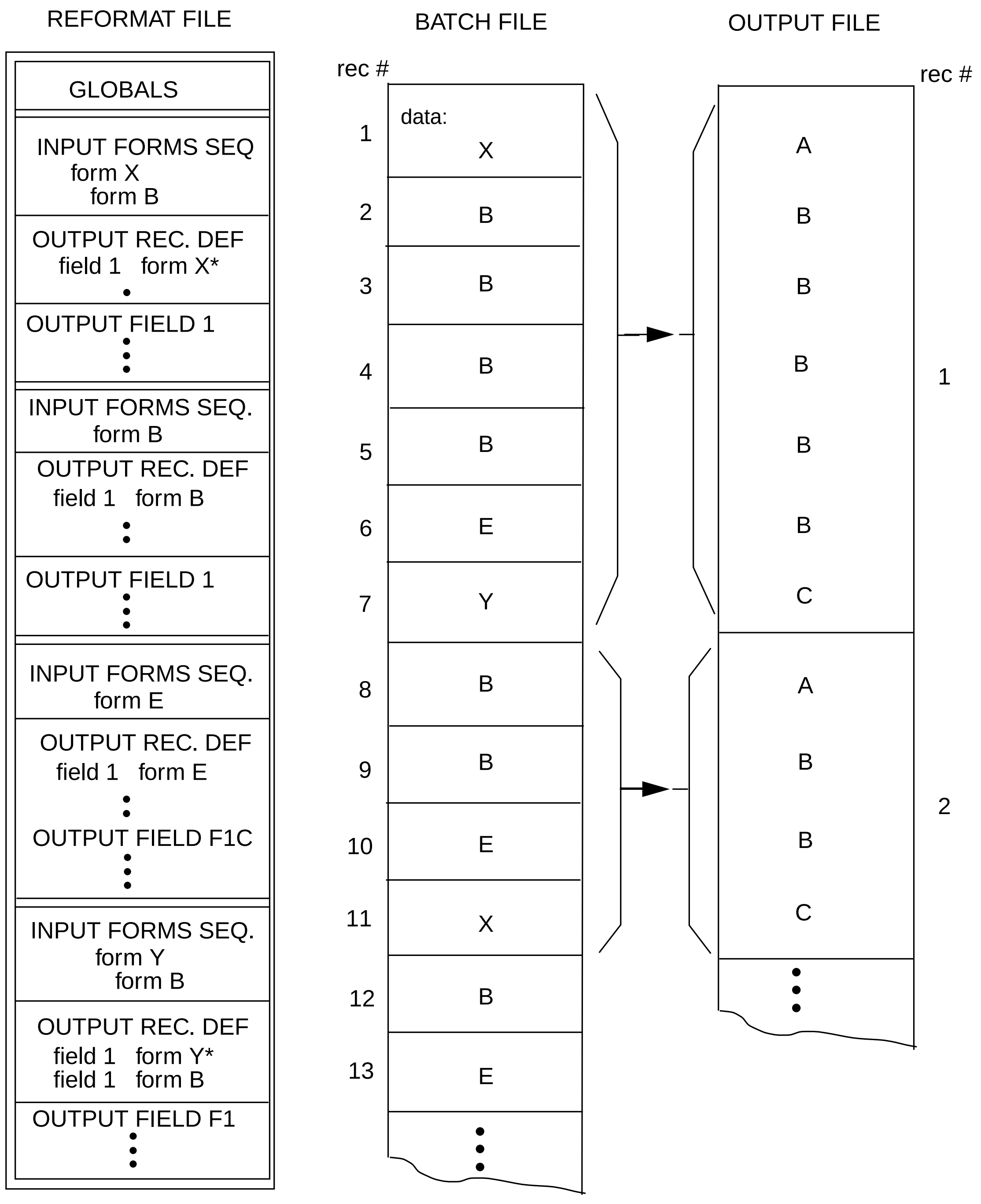
In Figure 5-6., form B is a repeating form that may occur a variable number of times. The reformat file in this example causes data entered on forms A, B, and C to be written to a single variable length output record. Note that if repeating form B causes so much data to be written to the output record that the maximum record length is exceeded, data (possibly significant) is truncated when REFORMAT is run.

Figure 5-6. Reformatting Data from Repeating Forms



Suppose data entered on one form can follow data entered on either of two different forms, and you want to generate separate records depending on the sequence in which the forms appear. In such a case, you can set up a reformat file as shown in Figure 5-7. Note that form B (a repeating form) appears as a reformat identifier and also as a succeeding form in other input forms sequences.

Figure 5-7. Reformatting Data Based on Form Sequences

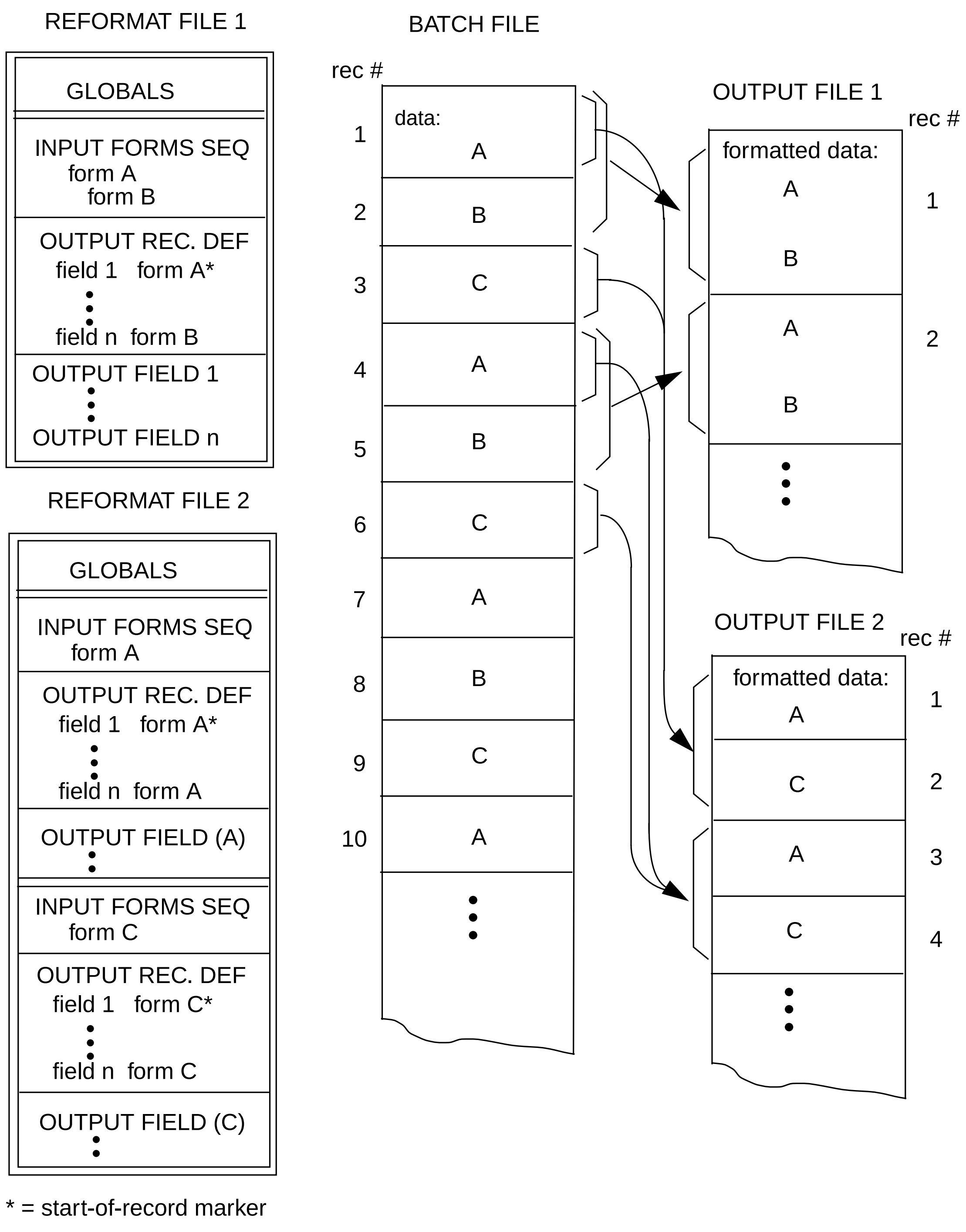


* = start-of-record marker

Separating Data from One Batch File into Several Output Files

A separate reformat file must be established for each different output file. In Figure 5-8., data entered on forms A and B is written as a single record to one output file. Data from forms A and C is written as two records to a second output file; data from form B is not written to output file 2.

Figure 5-8. Generating 2 Output Files from 1 Batch File



Using REFSPEC

You execute REFSPEC by entering the following command in response to the MPE colon prompt:

```
:RUN REFSPEC.PUB.SYS
```

Since REFSPEC runs entirely in block mode, your terminal is placed in block mode automatically.

Like FORMSPEC, REFSPEC prompts for information on menu screens. The information you enter on these menus defines how data in the batch file is to be reformatted and written to an output file. The first menu issued whenever you run REFSPEC is the Reformat File Menu.

If you specify a new reformat file, REFSPEC creates this file and then issues the Forms File Menu so you can specify the forms file through which the data to be reformatted was entered. If you specify an existing reformat file, it is already associated with a forms file so the Forms File Menu is not issued, and the Main Menu is the next menu.

NOTE If your file has a lockword, you *must* enter the lockword along with the file name, as *filename/lockword*. If you do not enter a lockword with the file name and one is required, your terminal hangs. This is because the MPE prompt requesting the lockword is in character mode and cannot be received while your terminal is in block mode. You can recover from this error by doing a hard reset followed by pressing **RETURN** at least four times, then pressing f8 to exit. How you perform a hard reset depends on what type of terminal you are using; consult your terminal manual for instructions.

Terminating REFSPEC

You can terminate operation of REFSPEC at any time by pressing EXIT to return to MPE control, indicated by the colon prompt (:).

When you next run REFSPEC after terminating and request the same reformat file, the Forms File Menu is skipped and the Main Menu is displayed on the screen. You may then select an option on the Main Menu or you can press MAIN/RESUME to display the Forms File Menu. If you specify a new reformat file on the Reformat File Menu, the next menu issued is the Forms File Menu.

Unexpected Program Interruption

As with FORMSPEC, if the program halts unexpectedly because of a terminal power failure or the user pressing **BREAK**, control returns to MPE. Refer to Appendix G for the steps to recover from such a situation. After you have recovered, you then press REFRESH to return to the REFSPEC Menu where you were interrupted. The menu is cleared to initial or previously entered values. To ensure against damage to the file, reenter the information on all menus pertaining to the reformat you were creating or modifying at the time of program interruption.

REFSPEC Function Keys

A set of seven function keys, similar to those used for forms design, are used to control execution of REFSPEC. These keys are described in Table 5-1.

Table 5-1. Key Definitions

Key	Key Action
PREV FORMAT f1	Go to previous Input Forms Menu (skipping any intervening menus).
NEXT FORMAT f2	Go to next Input Forms Menu (skipping any intervening menus).
f3	Not defined.
REFRESH f4	Redisplay current menu in its initial state before any specifications were entered, or existing specifications were modified.
PREV f5	Position to Previous Menu screen in sequence of menus.
NEXT f6	Position to Next Menu screen in sequence of menus.
MAIN/ RESUME f7	Request Main Menu or, if Main Menu displayed, return to menu displayed when Main was requested.
EXIT f8	Terminate REFSPEC and return to MPE control.

NOTE The function keys used by REFSPEC should not be confused with the function keys defined for ENTRY (see Figure 2-2). Although the two groups of keys are physically the same programmable keys, their functions differ significantly.

REFSPEC Menus

REFSPEC displays menu screens in a predetermined sequence. As each menu is displayed, enter the reformatting specifications you want in the reformat file, and then press **ENTER**. When **ENTER** is pressed, the specifications are written to the reformat file and the next menu in the sequence is issued. If you do not want to enter any specifications (on a menu such as the Globals Menu where defaults are supplied), simply press **NEXT** to display the next menu.

When the reformatting is completely defined, request the Main Menu by pressing **MAIN/RESUME**. You can then compile the reformat file so that it can be executed.

The REFSPEC menu screens are shown in Figure 5-9. in the order they are issued. First the Reformat File Menu requests the name of the reformat file. For a new reformat file, the next menu asks for the name of the forms file whose data is to be reformatted. For an existing reformat file, the Forms File Menu is skipped and the Main Menu displayed.

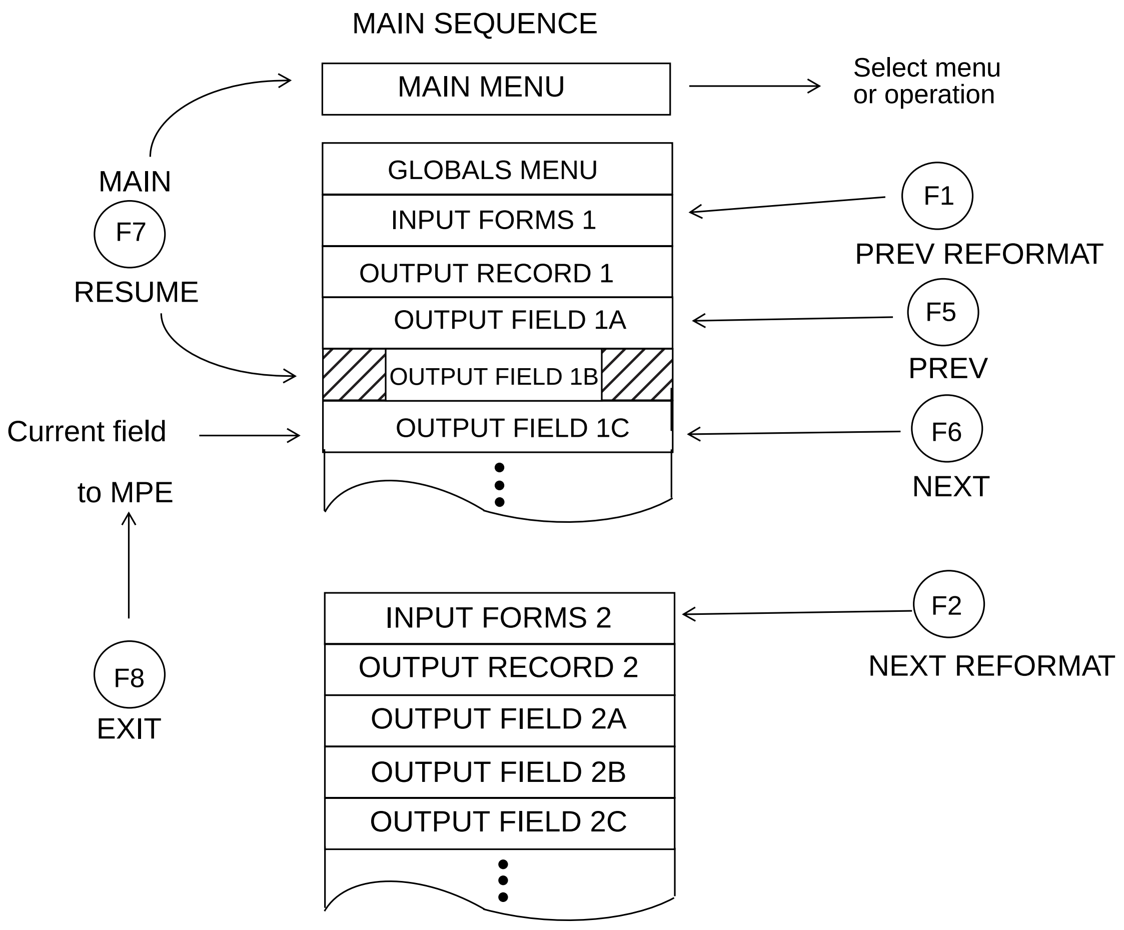
You can move forwards or backwards through the sequence of menus to locate a particular menu by pressing **NEXT** or **PREV**, respectively. You can skip an entire reformat specification by pressing **NEXT REFORMAT** or **PREV REFORMAT**. (A reformat consists of an input forms sequence and its associated output record definition.) These keys allow you to go directly to the next or previous Input Forms Menu. An alternate method is to request the Main Menu by pressing **MAIN/RESUME** and then selecting a particular menu.

Press **EXIT** to return to MPE control from REFSPEC.

Figure 5-9. illustrates the relationship between the menu definitions presented on the REFSPEC menu screens and some of the special function keys.

NOTE The number of Output Field Menus issued depends on the number of reformat fields defined in the preceding Output Record Menu. One Output Record Menu is issued for each Input Forms Menu. You may be able to specify reformatting with as few as one Input Forms Menu, or you may need as many as there are forms in the forms file.

Figure 5-9. Relation of REFSPEC Menus to Function Keys



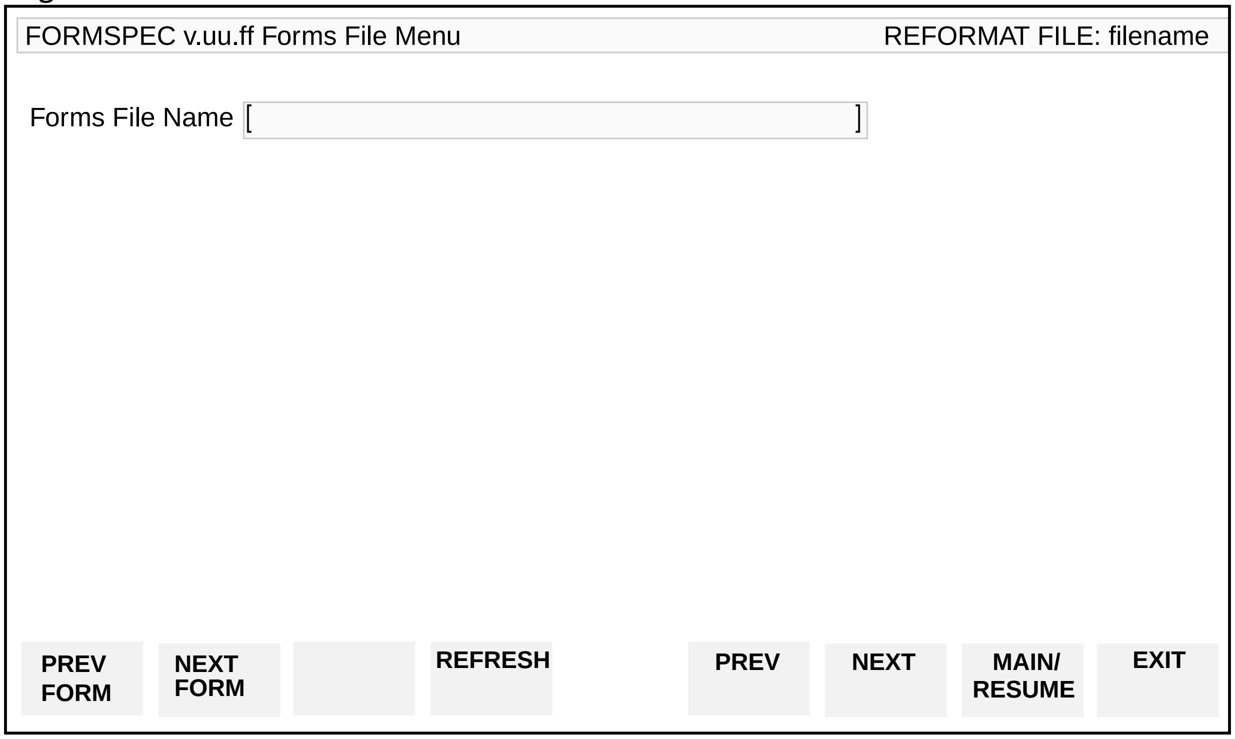
Forms File Menu

The Forms File Menu, shown in Figure 5-10., asks you to enter the name of an existing forms file. This is the file that was used to create the batch file (or files) to be reformatted. Only one forms file can be specified for each reformat file. The name of the data file may be a fully qualified MPE file name.

After you have identified the forms file, the Main Menu is issued.

NOTE In order to maintain (and browse or list) a forms file, the user must have exclusive write access to the file.

Figure 5-10. Forms File Menu



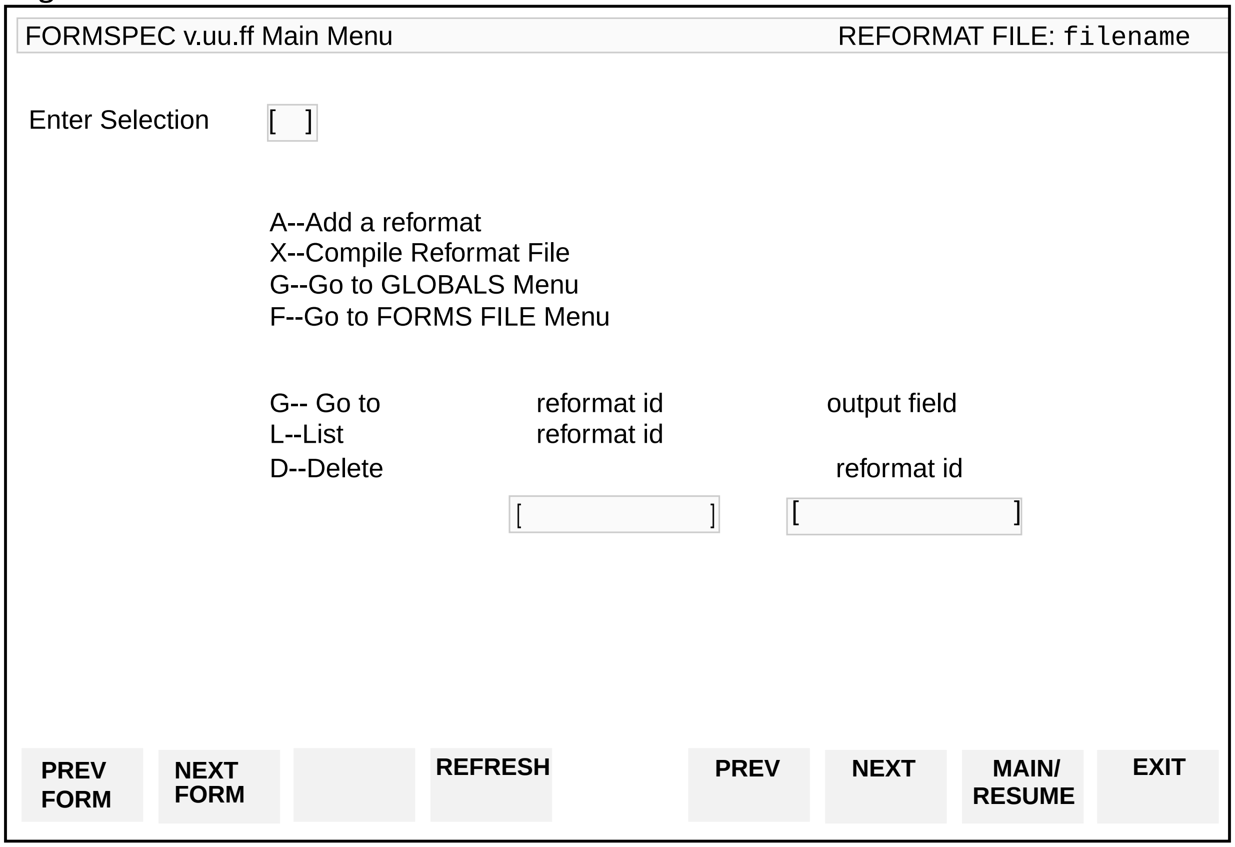
FORMSPEC v.uu.ff Forms File Menu REFORMAT FILE: filename

Forms File Name [ ]

Main Menu

The Main Menu, shown in Figure 5-11., is the main control menu for all REFSPEC operations. If the reformat file is new, you will usually select `Add a reformat`. REFSPEC then issues the menus that allow you to define the reformat file. If the reformat file already exists, you may enter any selection depending on what you want to do. You can compile the reformat file, add new reformats, select a particular menu in order to change it, purge reformats and so forth.

Figure 5-11. Main Menu



```
FORMSPEC v.uu.ff Main Menu                         REFORMAT FILE: filename

Enter Selection        [  ]

                       A--Add a reformat
                       X--Compile Reformat File
                       G--Go to GLOBALS Menu
                       F--Go to FORMS FILE Menu

                       G-- Go to           reformat id          output field
                       L--List             reformat id
                       D--Delete                                 reformat id
                                       [              ]    [              ]
```

Adding a Reformat

When you are creating a new reformat file, you will always add at least one reformat; if you are modifying an existing reformat file, you may want to add a reformat. New reformats are added to the end of the reformat file. To add a reformat, simply enter **A** in the Main Menu selection box. In response, REFSPEC displays the Input Forms Menu, shown in Figure 5-14.

Compiling the Reformat File

In order for REFORMAT to use the reformat file, the file must be compiled. You compile a reformat file by entering an **X** in the Main Menu selection box.

If the file has already been compiled, it is recompiled and the newly compiled version replaces the previous version. There is never more than one compiled version of a reformat file at a time. When you modify the source version, the compiled version is not affected.

Adding Global Specifications

Certain characteristics of the reformat definition are **global**; that is, they apply to the entire output file. For example, record length and the end of record separators are global definitions. (Refer to the Globals Menu description below.)

REFSPEC supplies default values for all global characteristics and, unless you want to change these defaults, you need never be concerned with global values. If you do want to specify nondefault global characteristics, enter **G** in the Main Menu selection box, but do not specify a reformat identifier or output field name. The Globals Menu, shown in Figure 5-13., is displayed so you can enter global specifications.

Changing the Forms File Name

Normally, you will not need to modify the forms file name once it has been specified. However, in case the forms file is renamed with the MPE `:RENAME` command, you can go to the Forms File Menu and rename the forms file to correspond to its new name. To do this, enter `F` in the Main Menu selection box and, when the Forms File Menu is displayed, type in the new forms file name.

Modifying Reformat Specifications

You can change a reformat specification at any time, either as you initially define the specification, or after the reformat file has been compiled. In either case, you locate the particular specification through the Main Menu selection box or with the function keys that control menu sequence.

Changing an Input Forms Sequence

Each input forms sequence consists of the form names associated with batch records to be reformatted and written to the output file. (Since each batch file record is associated with a form name, the list of form names is in effect a list of batch records.) The first form name in any input forms sequence is called the **reformat identifier**. Each reformat identifier is unique to the reformat file and is used to identify the input forms sequence.

If you want to modify an input forms sequence, enter **G** in the Main Menu selection box, and then specify the reformat identifier that identifies that sequence; or locate the input forms sequence you want with `PREV REFORMAT` or `NEXT REFORMAT`. The Input Forms Menu is displayed with the form names previously specified. You can then change any of these names, add new names, or delete existing names. When you have made the changes, press **ENTER**. You can change the reformat identifier exactly as you would change any other form name, but the changed identifier must still be unique to the reformat file.

Note that if you change an Input Forms Menu, you must validate the associated Output Record and you may have to validate the Output Field Menus. To do this, you request the applicable Output Record and output Field Menus and press **ENTER** for each. (If these menus were affected by your change to the Input Forms Menu, you must make the appropriate changes before pressing **ENTER**.) Until these menus are validated, REFSPEC prints a warning when they are displayed.

Changing an Output Record Definition

Every input forms sequence has an associated output record definition. The output record definition defines how the individual fields or portions of fields from the batch file are to be written to the output file. You can change some of these field specifications directly on the freeform Output Record Menu, and others on a particular Output Field Menu. The Output Field Menus can be located by entering **G** in the Main Menu selection box and then specifying a reformat id and output field name, or by using **NEXT** or **PREV**. An output record definition can be located only through these function keys.

The basic output record descriptions are entered and changed on the Output Record Menu. The output fields can be rearranged, new fields or constants added, or existing fields or constants deleted only on this menu. To change specific field reformatting specifications, you must go to the Output Field Menu for that field. (For details, refer to the Output Record Menu and Output Field Menu descriptions below.) After changing a reformat, you must compile the reformat file to enter these changes in the executable version of the file.

Deleting a Reformat

You can purge an entire reformat with its associated output record definition and field specifications. Enter **D** in the Main Menu selection box and then specify the reformat identifier that identifies the reformat to be deleted. The reformat specification is not physically removed from the reformat file, but it can no longer be referenced by REFORMAT. After deleting a reformat, you must compile the reformat file or the reformat will still be in the file.

Listing a Reformat

You can print a listing of the entire reformat file or only a single reformat on an off-line device (usually, the line printer). To list a single reformat, enter **L** in the Main Menu selection box and then specify the reformat identifier for the reformat you want listed. If you want to list all reformats in the reformat file, simply enter **L** in the selection box without specifying a reformat identifier. Figure 5-12. illustrates the listing of a reformat.

Resuming Design from Main Menu

If you requested the Main Menu by pressing **MAIN/RESUME** during reformat design, you can return to the menu you were in by pressing **MAIN/RESUME** again. Note that this is the same key in each case, but it acts differently in the Main Menu than it does in other menus.

You can also go to the previous menu or the next menu by pressing PREV or NEXT respectively. When requested from the Main Menu, the previous menu is the one preceding the menu you were in when you requested the Main Menu, and the next menu is that following the menu you were in. Similarly, you can request the previous or next reformat directly by pressing **PREV REFORMAT** or **NEXT REFORMAT**. The previous or next reformat is relative to the reformat you were designing when **MAIN/RESUME** was requested.

Figure 5-12. Reformat Listing

```
********************************************************************************
*                            REFSPEC Version v.uu.ff                           *
*                           MON, JUN 16. 1986,   5:24 PM                       *
*                                                                              *
*                             REF211.PUB.ACCTG                                 *
*                                                                        *
**************************************************************************
Reformat File Status
   Modified: TUE, DEC 31, 1985, 9:45 AM
   Compiled: TUE, DEC 31, 1985, 9:45 AM

Forms File: FORMS32.PUB.ACCTG

Output Record Format: F
Record Length: 80   (bytes)

Upshift? N
Convert to EBCDIC? N

Record Separator String:
Field Separator String:

REFORMAT FORM211

   Input Forms (in sequence):
     FORM211


   Output Record Definition

INPUT Field   Subst Len  Form Name           OUTPUT Field   Strt   Len   Strt
              Strt                                          Col           Rec
*******************************************************************************
“RUN 1.211; CHAR TO CHAR”                                                  *
FIELD1          1    6   FORM211              FIELD1        1     6        *
FIELD2          1    7   FORM211              FIELD2        7     7
FIELD3          1    8   FORM211              FIELD3        14    6
FIELD4          1    9   FORM211              FIELD4        22    9
FIELD5          1    10  FORM211              FIELD5        31    10
FIELD6          1    11  FORM211              FIELD6        41    11
FIELD7          1    12  FORM211              FIELD7        52    12
FIELD8          1    13  FORM211              FIELD8        64    13
FIELD9          1    14  FORM211              FIELD9        1     14       *
FIELD10         1    15  FORM211              FIELD10       15    16
FIELD11         1    16  FORM211              FIELD11       30    16
“MOVE TO LARGER FIELD”
DATA1          1    5   FORM211               DATA1         1     20       *
DATA2          1    6   FORM211               DATA2         21    20
DATA3          1    7   FORM211               DATA3         41    20
DATA4          1    8   FORM211               DATA4         61    20
DATA5          1    9   FORM211               DATA5         1     20       *
DATA6          1    10  FORM211               DATA6         21    25
DATA7          1    11  FORM211               DATA7         26    15
“MOVE TO SMALLER FIELD”
DATA8          1    12  FORM211               DATA8         1     11       *
DATA9          1    13  FORM211               DATA9         12    12
DATA10         1    14  FORM211               DATA10        24    2
DATA11         1    15  FORM211               DATA11        26    6
“MOVE TO PARTIAL FIELDS”
FIELD10         1    10  FORM211              TEMP10        1     10
FIELD11         4    10  FORM211              TEMP11        20    10
DATA10          1    10  FORM211              TEMP12        1     8        *
DATA11          3    10  FORM211              TEMP13        20    6
********************************************************************************
```

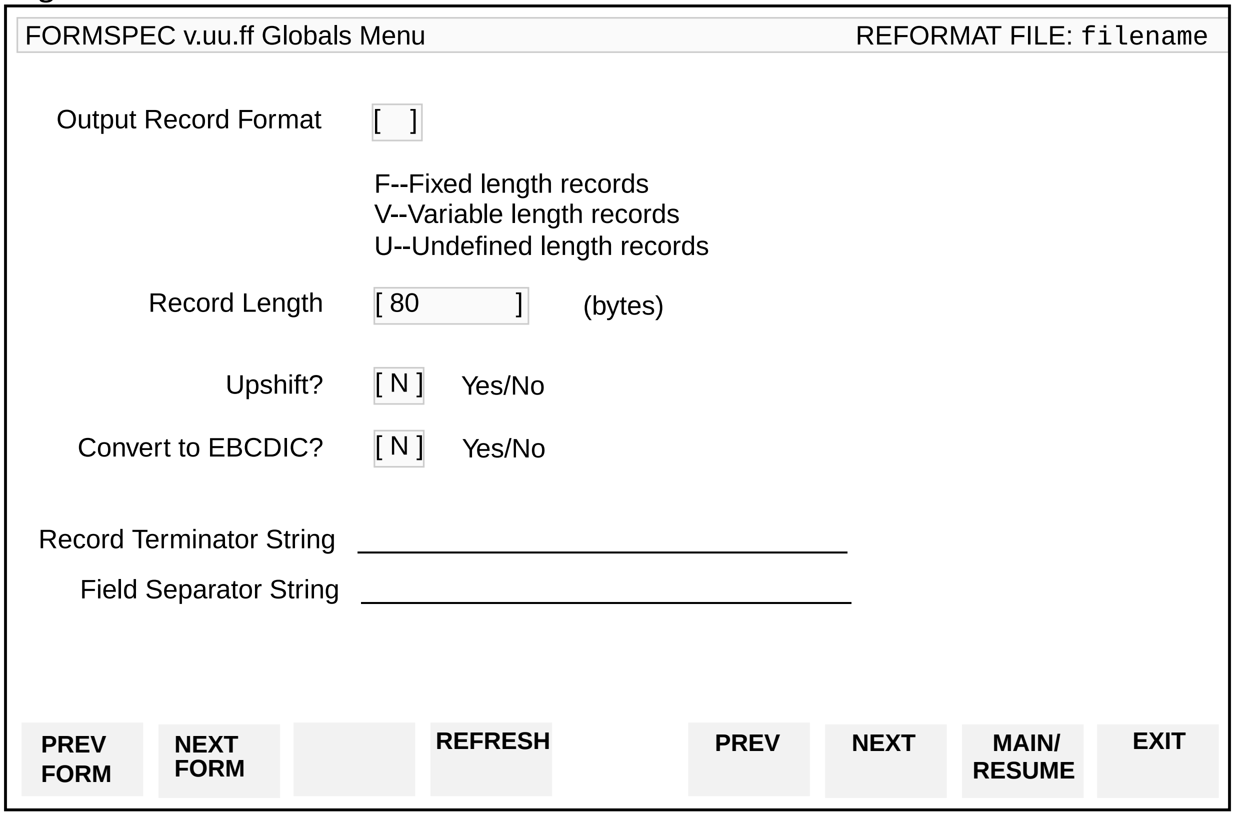
Figure 5-13. Reformat Listing (Continued)

```
REFSPEC VERSION v.uu.ff                                     MON, JUN 16, 1986, 5:24 AM
REFORMAT FILE: REF211.PUB.ACCTG
INPUT Field: FIELD1            Start: 1   Length: 6   Form: FORM211
OUTPUT Field: fIELD 1            Start: 1   Length:  6   Data Type: CHAR


STRIP:     All:                Leading:                 Trailing:
CHECK DIGIT:
JUSTIFY:
SIGN:                          PLUS SIGN?
FILL:       All:                Leading:                  Trailing:
                       .
                       .

INOUT Field: DATA11            Start: 3    Length: 10   Form: FORM211
OUTPUT Field: TEMP13           Start: 20   Length: 6    Data Type: CHAR

STRIP:     All:                Leading:                 Trailing:
CHECK DIGIT:
JUSTIFY:
SIGN:                          PLUS SIGN?
FILL:       All:                 Leading:                  Trailing:
                       .
                       .
                       .
REFORMAT FORM211
    Input Forms (in sequence):
        FoORM212

INPUT Field   Subst Len  Form Name           OUTPUT Field   Strt   Len   Strt
              Strt                                          Col           Rec
******************************************************************************
“RUN 1.212; NUMx FIELDS TO SAME NUMx”                                      *
NUM1             1      20   FORM212          N1            1      20      *
NUM2             1      20   FORM212          N2            30     20
NUM3             1      20   FORM212          N3            60     20
NUM4             1      20   FORM212          N4            2      20      *
NUM5             1      20   FORM212          N5            30     20
NUM6             1      20   FORM212          N6            60     20
NUM7             1      20   FORM212          N7            1      20      *
NUM8             1      20   FORM212          N8            30     20
NUM9             1      20   FORM212          N9            60     20
NUM10            1      20   FORM212          N10           1      20      *
NUM11            1      20   FORM211          N11           30     20
******************************************************************************

INOUT Field:  N1              Start: 3   Length: 20   Form: FORM212
OUTPUT Field: N1              Start: 20  Length: 20   Data Type: NUM

STRIP:       All:             Leading:                Trailing:
CHECK DIGIT:
JUSTIFY:
SIGN:                         PLUS SIGN?
FILL:         All:              Leading:                 Trailing:
                       .
                       .
                       .
```

Globals Menu

The Globals Menu, shown in Figure 5-13., requests information that applies to the total reformat process. All information entered on the Globals Menu relates to the output file produced when REFORMAT is run. Since only one output file is generated for each REFORMAT execution, this information appears only once in the reformat file.

Figure 5-13. Globals Menu



```
FORMSPEC v.uu.ff Globals Menu                         REFORMAT FILE: filename

     Output Record Format     [  ]

                              F--Fixed length records
                              V--Variable length records
                              U--Undefined length records

            Record Length     [ 80       ]    (bytes)

                Upshift?      [ N ]   Yes/No

       Convert to EBCDIC?     [ N ]   Yes/No

   Record Terminator String   ______________________________

      Field Separator String  ______________________________
```

When the Globals Menu is displayed, default values are shown for each option. The possible values and their defaults are listed below. You can change any of the values or you can keep the default values provided by REFSPEC. Press **ENTER** to record the values and request the next menu, the Input Forms Menu.

Fields

`Output Record Format` Specifies whether the output record is fixed, variable, or undefined in length. Enter:

- F - Fixed -length records
- V - Variable -length records
- U - Undefined-length records

Default = F (Fixed length)

`Record Length` A positive integer that specifies the total number of characters in the output record, including all fields and separators.

If the output records are fixed length and the total number of characters is less than this length, the record is padded with blanks at the end. If there are more characters in the field than will fit in the record, the record is written up to the specified length and the excess characters are discarded. In this case, a warning message is issued to the user who runs REFORMAT.

For variable length records, the actual record length is the sum of all the fields written to the record, including separators. In this case, the record length specified here is the maximum length allowed for any record.

Default = 80 characters.

`Upshift?` Indicates whether letters of the alphabet are to be shifted up to all uppercase letters when data is written to the output file. Specified as:

- Y — Shift letters to uppercase.
- N — Leave letters as entered by the user.

Default = N (do not upshift).

If nothing is specified, an error is returned.

`Convert to EBCDIC?` Indicate whether the output file is to be written in EBCDIC code rather than USASCII.

- Y - write output file in EBCDIC
- N - leave output file in USASCII

Default = N (do not convert).

`Record Terminator String` Indicates a character or string of characters to be appended to the end of every record. The record terminator may be specified as any of the following:

- Quoted String — Any USASCII characters, including blanks, enclosed within single or double quotes. For example: `“eof” or “ “ or ‘**’`.
- USASCII Code — Numeric equivalent to an USASCII character preceded by a dollar sign ($). Code may be any decimal number in range 0-127 (refer to Appendix C for decimal equivalent to USASCII code). For example: $34 is the numeric equivalent of quotation mark; $65 is the equivalent of the letter A.
- System Constant — The following system defined constants may be used as a terminator:
  - $LF — line feed
  - $CR — carriage return
  - $GS — group separator

— $US — unit separator

— $RS — record separator

Default = No terminator

If a record terminator is not specified, no special terminator is placed at the end of output records.

You can combine any of the above terminators into a single terminator by specifying them one after the other. For example: `"end" $LF $CR` or `$120 "! " $CR` or `"ABC" "DEF"`.

Such multiple terminators are concatenated together to form a single string that is inserted between fields. Note that blanks are optional except between quoted strings where a separating blank is required. If a blank does not separate quoted strings, the quote is included; for example: `"ABC""DEF"` becomes `ABC"DEF` in the record.

`Field Separator String` A user-defined value to be inserted between all consecutive fields in the output record. It will not appear after the last field in the record, or before or after a constant. If two consecutive fields are assigned specific column positions so that the second does not immediately follow the first, the field separator is placed after the first field. For example, suppose the field separator is, and the fields are defined as:

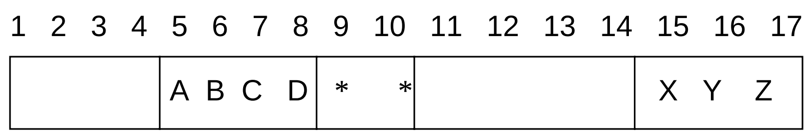
- FIELD 1 value = ABCD start column = 5
- FIELD 2 value = XYZ start column = 15

The output record for this example is shown below.

The separator may be a quoted string, a USASCII code, or a system constant, as described above for the Record Terminator.

Unless a separator is specified, fields in an output record are not separated, but are written as one continuous string of characters.

Default = No separator.

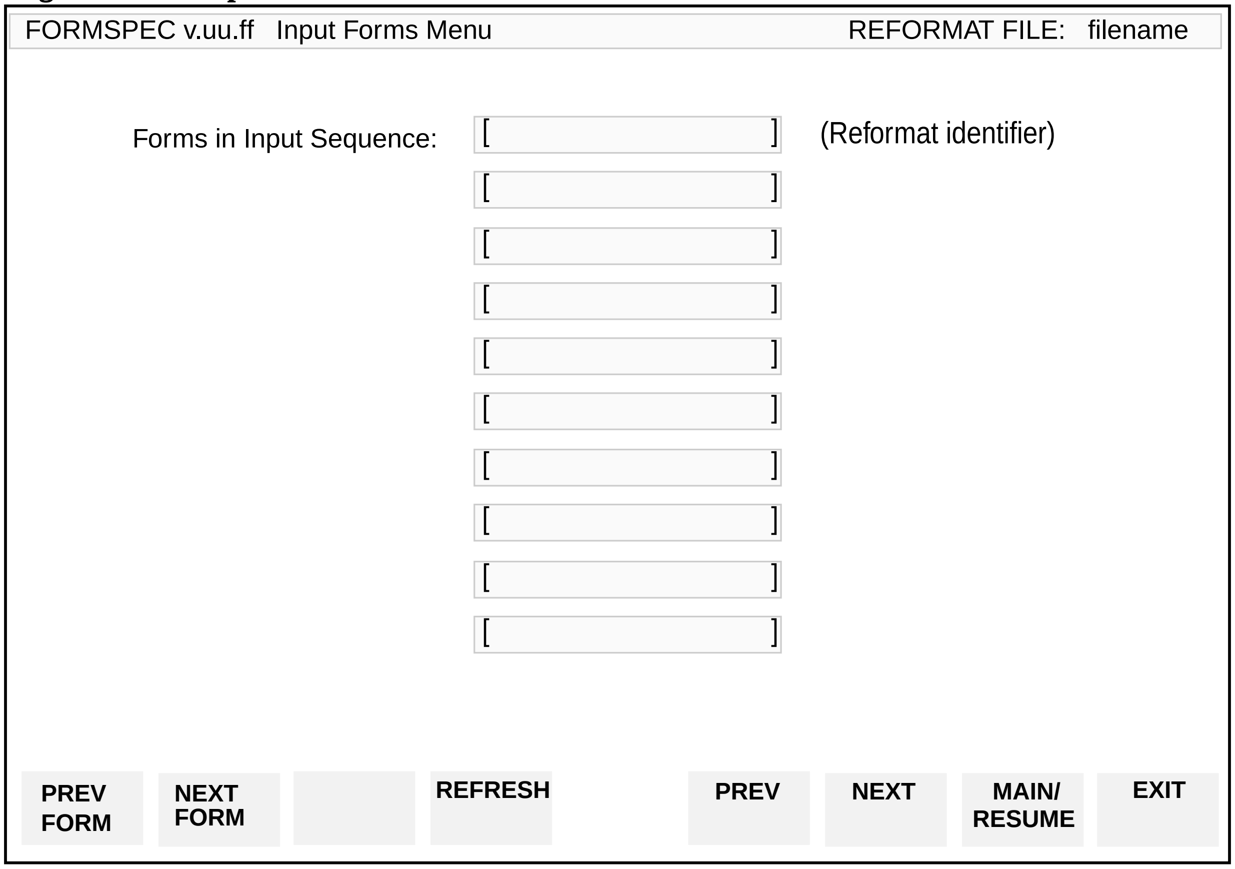


Input Forms Menu

The Input Forms Menu, shown in Figure 5-14., is used to specify one or more forms whose data is to be written to the output file. You can specify as many input forms sequences as there are forms in the forms file, or you can specify as few as one input forms sequence.

Each input forms sequence may contain a single form name or it may contain up to 10 form names. The first form in the sequence (or the only form) is the **reformat identifier**. All reformat identifiers must be unique to the reformat file. Subsequent form names in the sequence need not be unique.

Figure 5-14. Input Forms Menu


FORMSPEC v.uu.ff   Input Forms Menu   REFORMAT FILE: filename

Forms in Input Sequence: [ ] (Reformat identifier)
[ ]
[ ]
[ ]
[ ]
[ ]
[ ]
[ ]
[ ]
[ ]

Fields

`Forms in Input Sequence`

> The form names in any input forms sequence must be existing forms in the forms file named on the Forms File Menu for this reformat file. (Refer to Figure 5-10.)

Care must be taken when you specify a sequence of forms. The form names following the reformat identifier must be in exactly the same order as the forms appear in the batch file. (Each data record in the batch file is entered on a single form and the name of this form is included with the data in the batch file.) To illustrate, assume the following input forms sequence:

```
FORMA (reformat identifier)
FORMB
FORMD
```

And assume the first three records of data in the batch file were entered on forms in the sequence:

```
FORMA
FORMB
FORMC
```

The data entered on FORMA of this sequence will not be written to the output file. This is because of the method used by REFORMAT to match batch file records with input forms sequences in the reformat file.

REFORMAT:

- Reads records from the batch file in sequential order from the beginning. (Assume the first record was entered on FORMA.)
- Checks all reformat identifiers in the reformat file until it finds FORMA. (Assume FORMA is found as the identifier for the sequence FORMA, FORMB, FORMD.)
- Reads the next record from the batch file. (Assume the next record was entered on FORMB.)
- Checks form names in the input forms sequence following FORMA. (Assume the next form in the sequence is FORMB, and the check is satisfactory.)
- Reads the next record from the batch file. (Assume the next record was entered on FORMC.)
- Checks next form name in the input forms sequence after FORMB. (Assume this form is FORMD; the form names do not match.)
- Skip data record written on FORMA.

REFORMAT then takes the next record in the batch file, in this case, the record written on FORMB, and searches the entire reformat file for a reformat identifier, FORMB. If such a reformat identifier is found, it checks the rest of the input forms sequence (as described above) to make sure that all forms in the sequence match the forms on which the batch records following FORMB were entered.

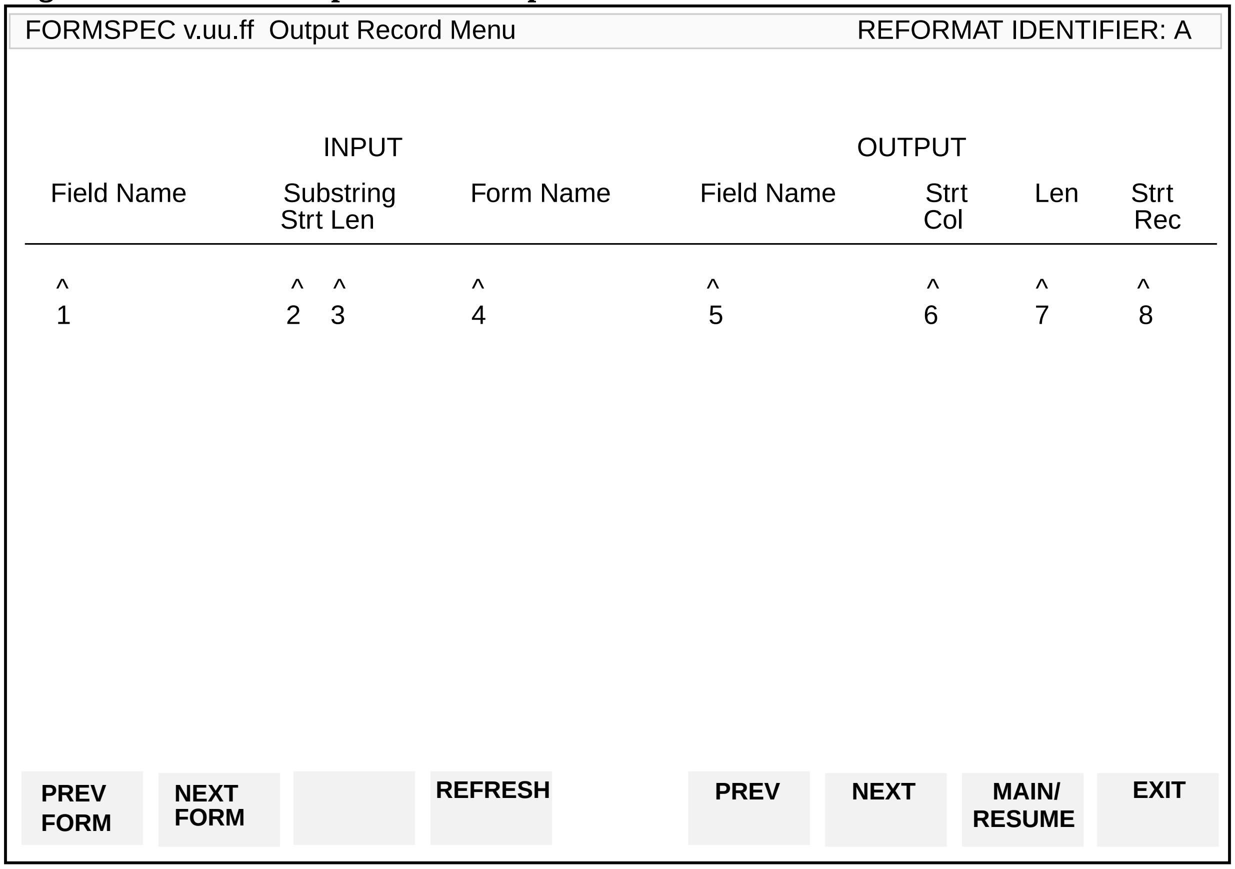
When a match is successful, each batch record is discarded as it is processed. If the match is unsuccessful, only the first batch record in a sequence is discarded.

Output Record Menu

An Output Record Menu, shown in Figure 5-15., is issued for every input forms sequence. It specifies which fields in the input forms are to be written to an output record. The output record definition may generate one output record, many output records, or only a portion of an output record. The order in which fields are specified is the order in which they are written to the output record.

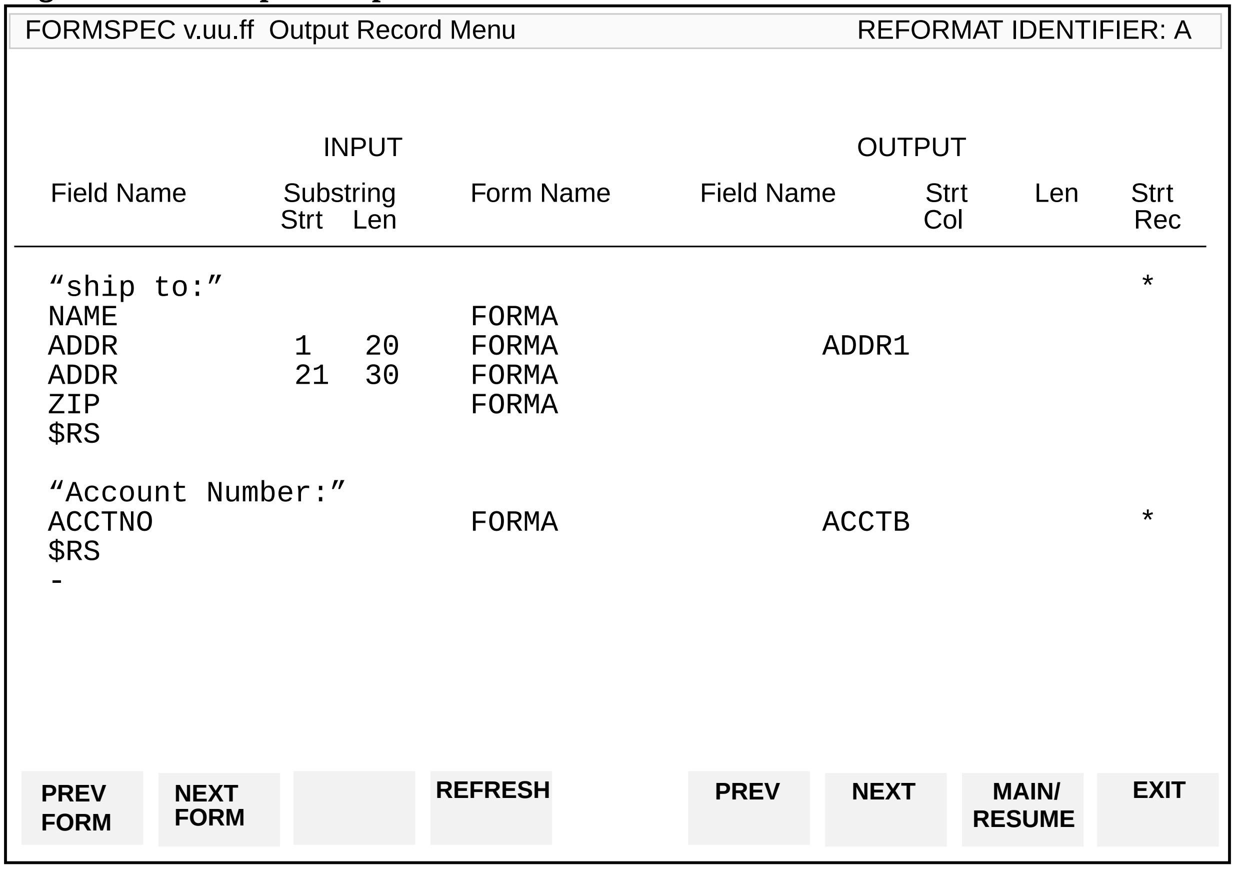
The Output Record Menu is not a true menu screen. Rather than selecting from predefined options, you enter the reformat fields in a freeform area on the screen. Once the output record definition is specified, you will have a complete list of the output record image for the forms specified in the corresponding input forms sequence. At a glance, you can see the order in which fields will appear and where each record begins. If only a portion of a field is to be written, this too is evident from the output record definition. As such, the output record definition can be thought of as the "screen image" of the reformat similar to the screen image generated for each form in the forms file.

Figure 5-15. An Example of an Output Record Menu


```
FORMSPEC v.uu.ff  Output Record Menu                    REFORMAT IDENTIFIER: A


                        INPUT                                  OUTPUT
  Field Name          Substring      Form Name      Field Name      Strt    Len    Strt
                      Strt Len                                      Col            Rec
  ---------------------------------------------------------------------------------------
  ^                   ^   ^          ^              ^               ^       ^      ^
  1                   2   3          4              5               6       7      8
```

When the Output Record Menu is displayed, the cursor is positioned to the first position in the first line in which you can enter an output record specification. Tabs are set so that when you press **TAB**, the cursor is moved to each subsequent position. Figure 5-16. illustrates an example of a completed Output Record Menu specification.

Figure 5-16. Sample Output Record Menu



```
FORMSPEC v.uu.ff  Output Record Menu                 REFORMAT IDENTIFIER: A

                        INPUT                                  OUTPUT
Field Name           Substring      Form Name        Field Name       Strt     Len     Strt
                     Strt   Len                                       Col              Rec
-----------------------------------------------------------------------------------------
"ship to:"                                                                              *
NAME                                FORMA
ADDR                 1      20      FORMA            ADDR1
ADDR                 21     30      FORMA
ZIP                                 FORMA
$RS

"Account Number:"
ACCTNO                              FORMA            ACCTB                              *
$RS
-
```

When the output record definition has been defined, press **ENTER**. A Field Menu is issued for each field name associated with an input field name in the output record definition.

Note that the only required entry is the field name; other specifications depend on the particular definition. You can specify a constant (literal string, system constant, or numeric equivalent of an USASCII character) instead of a field name. A constant may start anywhere on the line, but cannot extend past column 75.

NOTE Since the output record specifications are entered on a freeform area of the screen, you can type anywhere on the screen. You should be careful not to delete or overwrite lines accidentally. The field specifications should be entered in the positions to which the tabs are set.

The specifications under the heading `INPUT` identify the fields (or portions of fields) to be accessed from the batch file. The specifications under `OUTPUT` determine how the field is to be written to the output record.

You may enter as many specification lines as there is room in your terminal memory. (Check the terminal memory capacity in the reference manual for your terminal.)

Fields

`Field Name` (Required)

Identifies the field in the batch file to be written to the output file. The field name must identify an existing field in one of the forms specified in the preceding input sequence.

The field name need not be unique to the output definition. But, if it is not unique, a different output field name must be entered for each identical input field name.

`Substring Strt` (Optional)

If you want to retrieve a portion of an input field that is not at the beginning of the field, specify the first character position that you want written to the output field as the Subst Strt value. Character positions in a field are numbered from 1. For example, suppose you want only the last five characters of a 20-character field written to the output file, enter 16 under Subst Strt. Used in combination with the Len parameter, you can specify any group of consecutive characters in an input field.

Default = First character in field (1).

`Substring Len` (Optional)

Allows you to retrieve only a portion of an input field. For example, if you want only the first five characters in a field, leave the Subst Strt field blank and enter 5 under Len. By combining the Len specification with the Subst Strt specification, you can select any consecutive group of characters in the input field. For example, to select characters 5 through 14 of the input field, set Subst Strt to 5 and Len to 10.

Default = length of input field - Subst Strt + 1

`Form Name` (Optional)

Identifies the form in which the input field appears. This parameter can be included when a field with the same name appears in more than one form in the forms file. If omitted, and the field name is not unique, the first form in the input forms sequence in which the field appears is assumed.

Default = First form in which field appears.

`Field Name` (Optional)

In case the input field name is not unique, you can give the field (or portion of the field) a unique identifier in this position. Any legal 15-character identifier can be specified. (Note that any lowercase letters are shifted to all uppercase by REFSPEC.)

Default = Input field name.

`Strt Col` (Optional)

If you want the field to start in a particular column in the output record rather than immediately after the preceding field, you can specify a starting column. Column numbers in this case correspond to character (byte) positions in the output record. Columns are numbered starting with 1.

When you specify a new starting column, REFSPEC checks to make sure the starting column does not overwrite a previous field in the output record. If the starting column is not correctly specified, REFSPEC issues an error message.

One reason for changing the starting column is to ensure that a field starts on a word boundary. For example, suppose the first field to be written is five characters long. You can either increase the length of this field in the "Len" position to an even number of characters, or you can specify a new starting column for the next field.

Default = Consecutive fields with no intervening spaces.

`Len` (Optional)

You can specify the number of characters for the output field in this position. If you specify a length shorter than the input field, character type data is truncated when it is written to the output field; numeric data may not be converted if the field is too short.

If you plan to add a check digit to the field, you must specify the output field length as at least one character longer than the input field length.

Default = Input field length (or substring length).

`Strt Rec` (Optional)

In order to separate the output into records, you must specify the beginning of each record. Any character entered in the Strt Rec position of the Output Record Menu marks the associated field or constant as the beginning of a new record. For consistency, you should select a standard character, such as an asterisk.

If a starting record is not marked in the output record definition, the fields in the current definition follow the fields in the previous definition as part of the same record.

Default = Field is not first field.

Constant (Optional)

You can specify a constant instead of a field name. The constant can be any of the following:

- a character string (literal) enclosed within single or double quotes;
- the numeric equivalent of a USASCII character preceded by a dollar sign ($);
- one of the system-defined constants:

  — $CR carriage return

- — $LF line feed
- — $GS group separator
- — $US unit separator
- — $RS record separator

The following are acceptable constants:

```
"Part No." 'ABC' $65 $34"NAME"$34 $RS
```

Constants may not be specified on the same line as a field name. More than one constant can be specified on a line. When multiple constants are specified, they are concatenated into a single literal value. For example:

```
"Part No."
$GS $65.
```

Blanks are not significant except within quoted strings, where they must be included.

A constant or group of concatenated constants must not extend past column 75 of the screen. Apart from this restriction, they can appear anywhere on the line.

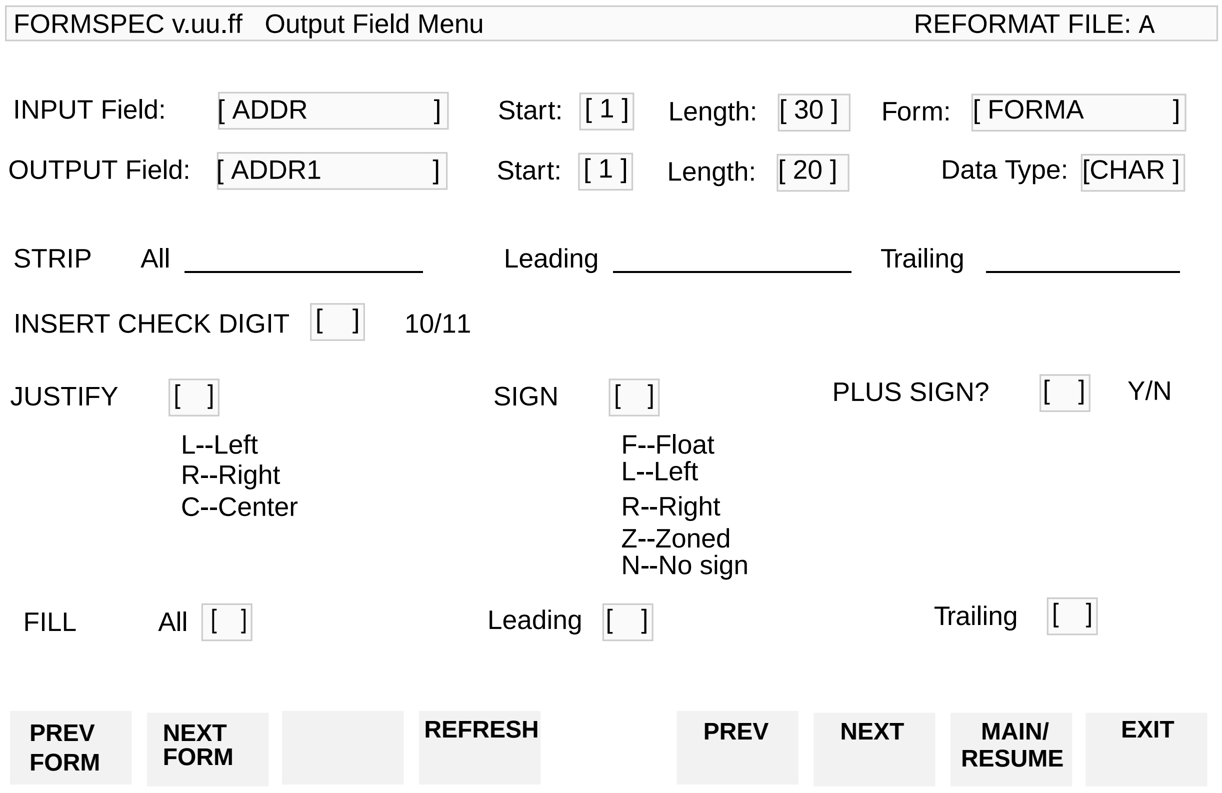
Default = No constant.

Output Field Menu

A separate Output Field Menu shown in Figure 5-17., is displayed for every input field named in the preceding Output Record Menu. The Output Field Menus are issued in the order the field names were entered in the output record definition. No Output Field Menu is issued for constants.

The Output Field Menu displays the information for a field entered on the preceding output record definition. In addition, it allows you to change the data type of a field from CHAR to any of the legal FORMSPEC data types. It also allows some formatting in addition to that performed by default when data is moved from the input field in the batch file to the output field in the output file.

Figure 5-17. Output Field Menu



```
FORMSPEC v.uu.ff   Output Field Menu                                REFORMAT FILE: A

INPUT Field:   [ ADDR      ]   Start: [ 1 ]   Length: [ 30 ]   Form:  [ FORMA     ]
OUTPUT Field:  [ ADDR1     ]   Start: [ 1 ]   Length: [ 20 ]   Data Type: [CHAR ]

STRIP     All ______________     Leading ______________     Trailing ___________

INSERT CHECK DIGIT [ ]   10/11

JUSTIFY   [ ]                    SIGN   [ ]                 PLUS SIGN?  [ ]   Y/N
          L--Left                       F--Float
          R--Right                      L--Left
          C--Center                     R--Right
                                        Z--Zoned
                                        N--No sign

FILL      All [ ]                Leading [ ]                      Trailing [ ]

PREV   NEXT              REFRESH        PREV    NEXT    MAIN/    EXIT
FORM   FORM                                             RESUME
```

The first line of the menu displays the input field values defined for this field on the preceding Output Record Menu. None of these values can be changed.

The second line displays the output field values defined for this field. Except for the data type, these values were established on the preceding Output Record Menu and cannot be changed on this menu. The data type of all fields to be written to the output file is defaulted to CHAR. This is done because only character type fields are written to the output record exactly as they were entered. If you want the field to be formatted to the format of a particular data type, you must change the data type specification on the Output Field Menu. (Refer to Table 5-2. on the following page for the standard formatting by data type.)

The remaining items in the Output Field Menu allow you to specify reformatting to be performed on the data item when it is moved to the output field. Note that these reformatting specifications cannot override the standard conversion and reformatting performed on the item based on the destination data type.

Data Type

You may change the output data type to a type other than CHAR. If, for example, the data type of the input item is NUM2, you can retain this data type by changing the output data type from CHAR to NUM2. This causes the value to be reformatted according to the standard rules for data type reformatting.

To illustrate, when a NUM2 item entered as `000123.12` is moved to a NUM2 field, leading zeros are replaced by blanks and it is right justified in the field. If moved to a CHAR field the value is moved to the output field exactly as entered. For another example, assume a date of type MDY is entered as `MAR 6, 1986`. If the output data type is left as CHAR, the date is written to the output field as `MAR 6, 1986`. If, on the other hand, the output data type is changed to MDY, the date is written as `03/06/86`. When you change a data type to one other than CHAR and different from the original data type, you must take care since not all data types are interchangeable, and those that are may require conversion of the data. Allowed changes are:

Any type to character	No conversion — data is left exactly as it was entered.
Numeric to numeric	Convert numeric value to conform to the destination type (DIGIT, NUM, NUMn, or IMPn).
Date to date	Convert date to *`dd/dd/dd`*; the exact order depends on the destination date type (MDY, DMY, or YMD).

Field Formatting

The remaining Output Field Menu specifications affect the actual data in the field. These specifications (`STRIP, JUSTIFY, SIGN, FILL,` and `CHECKDIGIT`) are performed in addition to the standard formatting performed when data is moved from an input (batch) field to an output field.

The standard formatting is described in Table 5-2. In general, the data is first converted to the destination data type, if necessary. Then all other formatting is performed. Note that the data is converted only if the output field type is specifically changed to a type other than CHAR.

Table 5-2. Standard Formatting by Data Types

Destination Data Type	**Standard Formatting**
`CHAR`	Conversion: none. Data movement: Move data to output field, character by character from leftmost character (including blanks) through last character (including blanks). If the data is shorter than the output field, pad with blanks on the right. If the data is longer than the output field, truncate on the right. `Assume a value with   length=10  ΔΔΔCHARΔΔΔ` `moved to CHAR field,  length=10  ΔΔΔCHARΔΔΔ` `to CHAR field,  length=15  ΔΔΔCHARΔΔΔΔΔΔΔΔ` `to CHAR field,  length=5   ΔΔΔCH`
`DATE`	Conversion and Movement: Convert entered date to *dd/dd/dd*, where the order of digits depends on the date type. A date is written to MDY type field as *mm/dd/yy*; to DMY type field as *dd/mm/yy*; to YMD field as *yy/mm/dd*. Write converted date, character by character, to the output field starting with the leftmost character. If the date is shorter than the field, fill with blanks on the right; if longer than the field, issue error message. `Assume an MDY date,  length=12  JAN 31, 1986` `moved to MDY field,  length=12  01/31/86ΔΔΔ` `to DMY field,  length=8   31/01/86` `to YMD field,  length=8   86/01/31` `to any date field,  length=7   ERROR`
`DIGIT`	Conversion and Movement: Strip any leading zeros. Right justify converted data in the output field. If the data is shorter than the field, pad it with blanks on the left. If the data is too long for the output field, strip any leading blanks, one at a time until the data fits. If, after all leading blanks are removed, the data does not fit, issue error message. `Assume an integer,     length=7   0012345` `moved to DIGIT field,  length=7   ΔΔ12345` `to DIGIT field,  length=10  ΔΔΔΔΔ12345` `to DIGIT field,  length=5   12345` `to DIGIT field,  length=4   ERROR`

Table 5-2. Standard Formatting by Data Types

Destination Data Type	Standard Formatting

`NUMn\or`
`\NUM`
	Conversion and Movement: Strip any commas, sign, or leading zeros. Float any minus sign to position preceding the first nonblank character. Right justify data in the output field. If the data is shorter than the field, pad with blanks on the left. If the data is too long, strip leading blanks one at a time until data fits. If, after all blanks are removed, the data is still too long, strip trailing fractional zeros, one by one. Then, if necessary, round fractional digits, one place at a time, until the value fits in the output field. (Note that rounding may change the value of the integer part.) If the value still does not fit, issue error message.
	
	`Assume value with        length=10    1,234.510`
	`moved to NUM3 field,     length=10    ΔΔ1234.510`
	`      to NUM3 field,     length=ll    ΔΔΔ1234.510`
	`      to NUM3 field,     length=6     1234.5`
	`      to NUM3 field,     length=4     1235`
	`      to NUM3 field,     length=3     ERROR`
	
	`Assume signed value,     length=10    +1234.510Δ`
	`moved to NUM2 field,     length=10    ΔΔΔ1234.51`
	`      to NUM2 field,     length=3     ERROR`
	
	`Assume signed value,     length=7     -12.10Δ`
	`moved to NUM2 field,     length=7     Δ-12.10`
	`      to NUM2 field,     length=3     -12`
	`        to NUM2 field,     length=2     ERROR`

`IMPn`	Conversion and Movement: Strip any decimal indicator, commas, sign, or leading zeros. Float any minus sign to the position preceding the first nonblank digit. Right justify data in the output field. If the data is shorter than the field, pad with blanks on the left. If the data is too long, strip leading blanks, one by one. If the data does not fit and only the fractional part remains, strip leading fractional zeros. (Trailing fractional zeros are never stripped from an IMPn field.) If the data still does not fit, issue error message.
	
	`Assume a value,       length=10   -123.0120ΔΔ`
	`moved to IMP4 field,  length=10   ΔΔΔ-1230120`
	`      to IMP4 field,  length=8    -1230120`
	`      to IMP4 field,  length=7    ERROR`
	
	`Assume a value,       length=7    .0120ΔΔ`
	`      to IMP4 field,  length=3    120`
	`        to IMP4 field,  length=2    ERROR`

REFSPEC performs all formatting (user-defined and standard) in the following order:

1. Convert data to destination data type (unless type is CHAR). If numeric type data does not fit in the output field after conversion, the output field is set to all blanks,
2. `STRIP` (user-specified)
3. Move data to output field, justified left, and inserting check digit if specified, and padding with blanks as needed. If character type data does not fit, it is truncated on the right. If a data field does not fit, the output field is set to all blanks.

   Perform any `CHECKDIGIT` (user-specified) and/or `JUSTIFY` (user-specified).
4. `SIGN` (user-specified)
5. `FILL` (user-specified)

Note that step 3 includes data type formatting with data movement and user-specified justification and check digit insertion. These steps are performed simultaneously so that significant data is not lost due to justification. For example, if you justify data in a field to the left, the data is justified before movement; but if you justify data to the right, the justification is performed after the data is moved.

Any changes (truncation, conversion, and so forth) made to the data in the output field do not affect the original data in the input field.

STRIP

This option lets you remove a particular character or group of characters from data entered in the field. The following three options are provided:

`STRIP ALL` *`characters`*

Strips each occurrence of each specified character. The remaining characters are shifted left to fill the space created by stripping the specified characters.

`STRIP LEADING` *`characters`*

Strips each occurrence of each specified character only if it appears before any other nonblank characters in the field. Stripped characters are replaced by blanks. It is meaningless to strip leading blanks; if you want to shift data left, use `JUSTIFY LEFT`.

`STRIP TRAILING` *`characters`*

Strips each occurrence of each specified character only if it appears after all other nonblank characters. Stripped characters are replaced by blanks.

In Table 5-3. are examples using `STRIP` where both the input field and the output field are 12 characters long and are the same data type; a blank is shown by the character Δ.

Table 5-3. Examples Using STRIP

Data Type	Input Value	Specification	Output Value
CHAR	ΔACBAXCBYZBB	none STRIP ALL "ABC" STRIP LEADING "ABC" STRIP TRAILING "ABC"	ΔACBAXCBYZBB ΔXYZΔΔΔΔΔΔΔΔ ΔΔΔΔΔXCBYZBB ΔACBAXCBYZΔΔ
DIGIT	Δ234567ΔΔΔΔ	none STRIP ALL "35" STRIP LEADING "0" STRIP TRAILING "7"	ΔΔΔΔΔ1234567 ΔΔΔΔΔ12467ΔΔΔ ΔΔΔΔΔ12345.67 ΔΔΔΔΔ3456Δ
NUM2	Δ12,345.67ΔΔ	none STRIP ALL "35" STRIP LEADING "01" STRIP TRAILING "7"	ΔΔΔ-12345.67 ΔΔΔ-124.67ΔΔ ΔΔΔΔ-2345.67 ΔΔΔ-12345.6Δ
IMP2	Δ12,34567ΔΔΔ	none STRIP ALL "35" STRIP LEADING "01" STRIP TRAILING "7"	ΔΔΔΔ-1234567 ΔΔΔΔ-12467ΔΔ ΔΔΔΔΔ-234567 ΔΔΔΔ-123456Δ

Note that in the last IMP2 example, stripping the last character moves the implied decimal to the position between 4 and 5.

JUSTIFY

This specification lets you move data to the right or left boundary of the output field or to center it in the field. These three options are specified as:

`JUSTIFY RIGHT`

Moves the data to the right until the last character in the output field is nonblank, padding with blanks on the left as the data is moved.

`JUSTIFY LEFT`

Moves data to the left until the first character in the output field is nonblank, padding with blanks on the right as the data is moved.

`JUSTIFY CENTER`

Positions the data so that there is an equal number of blanks to the right and left of the nonblank data. If the total number of blanks in the field is not even, the extra blank is on the right.

If JUSTIFY is not specified, numeric type data is justified right and date type data is justified left. No justification is performed on character type data. In Table 5-3., the examples illustrate the three types of justification. It is assumed the input and output data types are the same.

Table 5-4. Examples of Three Types of Justification

Data Type	Input Value	Specification	Output Value
CHAR	ΔABCDEFΔΔ	JUSTIFY R JUSTIFY L JUSTIFY C	ΔΔΔΔABCDEF ABCDEFΔΔΔΔ ΔΔABCDEF
DATE	1/30/86ΔΔΔ	none JUSTIFY R JUSTIFY C	1/30/86ΔΔ ΔΔ01/30/86 Δ01/30/86Δ
DIGIT NUM or IMP	ΔΔ123456ΔΔ	none JUSTIFY L JUSTIFY C	ΔΔΔΔ123456 123456ΔΔΔΔ ΔΔ123456ΔΔ

Sign

With the SIGN specification, you can indicate where you want either a plus or minus sign placed in the output field. Any plus sign is removed automatically when a number is moved to the output field unless you specifically request that it be included with the PLUS option of SIGN. SIGN has the following options:

SIGN LEFT	The sign is placed in the first character position of the field. If this position contains a blank, the sign replaces it. Otherwise, data may be moved to the right of a leading zero stripped to accommodate the sign. If needed, a fractional digit is rounded. If the sign still does not fit, an error occurs and the field is set to blanks.
SIGN RIGHT	The sign is placed in the last character position in the field. As with SIGN LEFT, every attempt is made to fit the sign in the field, but if it results in too many characters, an error occurs and the field is set to blanks.
SIGN FLOAT	The sign is placed immediately preceding the first nonblank digit in the field. As with SIGN LEFT a leading zero may need to be stripped or a fractional digit rounded to accommodate the sign. If it still does not fit, an error occurs and the field is set to blanks.
SIGN ZONE	The sign is represented as an "overpunch" character in the last digit of the field. (See Table 5-2. for the value of a zoned sign.) No movement of data is required for this option.
NO SIGN	Any sign in the input field is stripped from the field when it is moved to the output record.
PLUS	All the preceding specifications apply to either a minus or a plus sign. The default is to insert only minus signs. If you want to retain a plus sign, you must enter Y in the PLUS option as well as indicate where you want the sign positioned.

If SIGN is not specified, a minus sign is floated for numeric type data and any plus sign is stripped; character type data is written exactly as entered.

Table 5-5. Correspondence Between Signed Digits and Characters

Positive Values	Positive Values	Negative Values	Negative Values
Signed Digit	Character	Signed Digit	Character
+0	{	-0	}
+1	A	-1	J
+2	B	-2	K
+3	C	-3	L
+4	D	-4	M
+5	E	-5	N
+6	F	-6	O
+7	G	-7	P
+8	H	-8	Q
+9	I	-9	R

In Table 5-6. are examples that illustrate the use of the SIGN option. The input and output fields are assumed to be the same data type.

Table 5-6. Example Using the SIGN Option

Data Type	**Input Value**	**Spec.**	**Output Value**
NUMO	-123456 +123456 -123456 -1234567 +1234567 -1234567	none none NO SIGN SIGN ZONE SIGN ZONE, PLUS SIGN RIGHT	ΔΔΔ-123456 ΔΔΔΔ123456 ΔΔΔΔ123456 ΔΔΔ123456P ΔΔΔ123456G ΔΔ1234567-

Fill

This specification allows you to replace leading, trailing, or all blanks in a field by a particular character. The specified character may be any printable character. It may not be a nonprinting control character, nor may it be more than one character. If FILL is not specified, no default replacement is made.

FILL ALL *character*

Replaces all blanks in the data with the specified character.

FILL LEADING *character*

Replaces all leading blanks with the specified character.

FILL TRAILING *character*

Replaces all trailing blanks with the specified character.

Default = none

You can specify both FILL LEADING and FILL TRAILING with no redundancy. But, you must not specify either FILL LEADING or FILL TRAILING with FILL ALL. Since the FILL ALL fills all the blanks, any other FILL specification for the field is diagnosed as an error. The examples in Table 5-7. illustrate use of the FILL option. The input and output data types are the same.

Table 5-7. Examples Using the FILL Option

Data Type	Input Value	Specification	Output Value
CHAR	ΔΔABCΔDEFΔΔ	FILL ALL* FILL LEADING* FILL TRAILING*	**ABC*DEF** **ABCΔDEFΔΔ ΔΔABCΔDEF**
NUM2	ΔΔΔΔΔ123.75	FILL LEADING*	*****123.75

FILL LEADING and FILL ALL are not sensitive to the sign of a signed number. For example, the first set of specifications in Table 5-8. produces a meaningless result because the sign is floated before the field is filled with zeros; the second set of specifications produces the desired result.

Table 5-8. Example Using the FILL LEADING and FILL ALL Option

Input Value	Specification	Output Value
-ΔΔ999	FLOAT SIGN FILL LEADING 0	000-999
-ΔΔΔ999	SIGN LEFT FILL ALL 0	-ΔΔΔ999 -000999

Check Digit

You can request that a check digit be added to the end of any digit or alphabetic value by entering 10 or 11 in the CHECKDIGIT option of the Output Field Menu. The check digit is calculated by modulo 10 or modulo 11 depending on which you specify.

Check digits can be added only to fields that are type DIG or type CHAR and that contain only digits or letters of the alphabet.

If ADD CHECK DIGIT is not specified as 10 or 11, no check digit is added.

NOTE This specification adds a check digit. If you want to verify a number that contains a check digit, this can be requested in the original form design using FORMSPEC (refer to Chapter 4).

If the data is right justified, the nonblank digits in the field are moved left one character position to make room for the check digit. Note that when a check digit is to be added to a field, the length of the output field must be specified as at least one character longer than the input field. (Field length is increased on the Output Record Menu, not the Output Field Menu.) Refer to Appendix D for a description of how check digits are calculated if modulo

10 or modulo 11 is specified.

Examples:

`CHECKDIGIT 10` Calculate check digit according to modulo 10 formula and add it following rightmost nonblank digit.

`CHECKDIGIT 11` Calculate check digit according to modulo 11 formula and add it following rightmost nonblank digit.

If the product generated by a `CHECKDIGIT 11` calculation evaluates to 10, this is considered invalid and the following message is issued when REFORMAT is executed:

```
"Check digit is invalid for modulus 11 calculation."
```

If the product generated by a `CHECKDIGIT 11` calculation evaluates to 11, a zero is appended to the basic number.

Running REFORMAT

Once the reformat specifications have been defined and are stored in a compiled reformat file, you can run REFORMAT to actually reformat the data in the batch file and write it to the output file.

REFORMAT is not an interactive program. That is, it does not prompt for any information. It is usually run as a batch job, though it can be run from a terminal directly or as a "streamed" job.

REFORMAT needs the names of three files: the "reformat" file containing the reformat specifications, the "batch" file containing the data to be reformatted, and the "output" file to which the reformatted data is written. To specify these files, use MPE `:FILE` commands before running REFORMAT, as follows:

```
:FILE REFENTITY =reformatfile
:FILE BATCH=batchfile
:FILE OUTENTITY =outputfile
```

For example, assuming a reformat file named `REF1`, a batch file named `BAT1`, and an output file named `OUT1`, the following commands are needed to run REFORMAT:

```
:FILE REFENTITY =REF1
:FILE BATCH=BAT1
:FILE OUTENTITY =OUT1
:RUN REFORMAT.PUB.SYS
```

These file equations show the required files for running REFORMAT. You will also probably want to list the reformatted data and, if you are running REFORMAT from a terminal, you will want error messages listed on the line printer rather than on the terminal screen.

A list of the reformatted data is very important when you run REFORMAT the first time, so you can see whether your reformat specifications are doing what you expect. To list reformatted data, include the file `TESTLIST` in the file equation that sends the listing to the line printer:

```
FILE TESTLIST;DEV=LP
```

Another important file, particularly when you are running a session, is the error message file. This file `REFLIST` normally is written to `$STDLIST`. Since `$STDLIST` in a session is the terminal, you may want to equate this file to the line printer with the command:

```
:FILE REFLIST;DEV=LP
```

Concatenating Batch Files

Data from a batch file can be appended to data from another batch file in an existing output file by issuing the following commands:

```
:FILE REFENTITY =reformatfilename
:FILE OUTENTITY =outputfilename
:FILE BATCH=firstbatchfilename
:RUN REFORMAT.PUB.SYS
```

These commands will run REFORMAT using the three files specified in the `:FILE` commands. Next, issue the following commands:

```
:FILE OUTENTITY =outputfilename;ACC=APPEND
:FILE BATCH=secondbatchfilename
:RUN REFORMAT.PUB.SYS
```

The outputfile now contains the reformatted data from the first and second batch files.

Using a User-Defined Command

You may want to combine the three `:FILE` commands required to run REFORMAT into one user-defined command (UDC). You enter a UDC through a text editor, as follows:

```
REFORMAT REFSPECS, DATA, OUTPUT=$STDLIST
FILE REFENTITY =!REFSPECS
FILE BATCH=!DATA
FILE OUTENTITY =!OUTPUT
FILE REFLIST;DEV=LP
FILE TESTLIST;DEV=LP
RUN REFORMAT.PUB.SYS
```

Use the SETCATALOG command to set the UDC.

The UDC definition is recorded by the MPE :SETCATALOG command. Now, all you need to do in order to run REFORMAT, is to enter the following command:

```
:REFORMAT reformatfilename, batchfilename, outputfilename
```

Suppose your reformat file is named REF1, your batch file is named BAT1, and your output file is OUT1, run REFORMAT as follows:

```
:REFORMAT REF1, BAT1, OUT1
```

Streaming REFORMAT

You may also want to stream REFORMAT. You must first create a text file of the commands to run the job through a text editor, and then issue the MPE :STREAM command to actually run the program. For example:

```
!JOB USER.ACCOUNT
!FILE REFENTITY =REF1
!FILE BATCH=BAT1
!FILE OUTENTITY =OUT1
!FILE TESTLIST;DEV=LP
!FILE REFLIST;DEV=LP
!RUN REFORMAT.PUB.SYS
!EOJ
```

To stream this job, assuming you named the text file REFSTREM, use the following command:

```
:STREAM REFSTREM
```

In many cases, a streamed job is part of a group of streamed jobs. When REFORMAT is part of a series of program executions, you should precede the `RUN REFORMAT` command by a MPE `:CONTINUE` command. Otherwise, any fatal error in REFORMAT (such as an inappropriate file name) prevents subsequent programs from executing. For example:

```
!JOB
!FILE ...
!FILE ...
!FILE ...
!CONTINUE
!RUN REFORMAT.PUB.SYS
!SPLGO MYPROG
!EOJ
  .
  .
  .
```

6 USING VPLUS INTRINSICS

A set of callable intrinsics is provided by VPLUS. Table 6-1. lists the VPLUS intrinsics in alphabetic order and summarizes their functions. A full description of each intrinsic appears later in this section. They can be used by any application, either for data entry or for other terminal related applications. The VPLUS intrinsics manage the interface between an application, the terminal, a forms file, the entered data, and, optionally, the batch file to which entered data may be written. These intrinsics are used by the VPLUS Data Entry Program (ENTRY) to control data entry.

Appendix A contains listings in different programming languages of sample applications. These listings provide useful examples of how to use the VPLUS intrinsics.

Multipurpose

The VPLUS intrinsics can be called by any application that displays forms on a terminal supported by VPLUS. With the exception of the batch file management intrinsics used specifically for data entry, the VPLUS intrinsics can be used in conjunction with FORMSPEC to allow for data base inquiries and updates via the application, as well as any other operation that results in the display or collection of data through a terminal form. Such applications can also make use of the capabilities that VPLUS provides for validating input data.

Multilanguage

The VPLUS intrinsics can be called from programs written in any of the supported programming languages listed in Appendix A. Additionally, some languages (also listed in Appendix A) provide the programmer with a special interface with terminals and forms, as described in their respective reference manuals. With these languages, the programmer does not call the VPLUS intrinsics directly. Instead, the programmer specifies the statements appropriate to the special interface. Each of the supported programming languages calls the VPLUS intrinsics with the same parameters, and these parameters are essentially the same type and size. So that the intrinsics can adjust to any peculiarities of the calling programming language, one input sub-parameter specifies the language of the calling program.

Error Detection

If a system or program error causes a VPLUS intrinsic to fail, an error code is returned to the calling program. Once an error has been detected, subsequent intrinsic calls do not perform any functions until the program detects the error and performs its own error routine. In order for processing to continue after an error is detected, the calling program must reset the error argument. This method of error handling means that intrinsic errors do not cause unexpected program termination.

Table 6-1. Summary of VPLUS Intrinsics

INTRINSIC	FUNCTION
VCHANGEFIELD	Changes field attributes for specified fields at run-time.
VCLOSEBATCH	Closes batch file.
VCLOSEFORMF	Closes forms file.
VCLOSETERM	Closes terminal file.
VERRMSG	Returns message associated with error code.
VFIELDEDITS	Performs field phase processing specifications.
VFINISHFORM	Performs final phase processing specified for form.
VGETBUFFER	Copies contents of data buffer into application.
VGETFIELD	Copies field contents from data buffer into application.
VGETFIELDINFO	Returns field information.
VGETFILEINFO	Returns forms file information.
VGETFORMINFO	Returns form information.
VGETKEYLABELS	Returns global or form function key labels.
VGETLANG	Returns the native language ID of the forms file being executed.
VGETNEXTFORM	Reads next form into form definition area of memory; window and data buffer are not affected.
VGETtype	Copies field contents from data buffer to application, converting data to specified type.
VINITFORM	Sets data buffer to initial values for form.
VLOADFORMS	Loads the specified forms into terminal local form storage memory.
VOPENBATCH	Opens batch file for processing.
VOPENFORMF	Opens forms file for processing.
VOPENTERM	Opens terminal file for processing.
VPLACECURSOR	Positions the cursor at a specified field after a form is displayed.

Table 6-1. Summary of VPLUS Intrinsics

INTRINSIC	FUNCTION
VPOSTBATCH	Updates end of file mark in batch file after last record referenced.
VPRINTFORM	Prints current form and data buffer on off-line list device.
VPRINTSCREEN	Prints entire contents of screen on off-line list device.
VPUTBUFFER	Copies data from application to data buffer.
VPUTFIELD	Copies data from application to field in data buffer.
VPUJTtype	Copies data of specified type from application to data buffer, converting data to external format.
VPUTWINDOW	Copies message from application to window area in memory for later display.
VREADBATCH	Reads record from batch file into data buffer.
VREADFIELDS	Reads input from terminal into data buffer.
VSETERROR	Sets error flag for data field in error and copies error message to window area.
VSETKEYLABEL	Temporarily sets a new label for a function key.
VSETKEYLABELS	Temporarily sets new labels for function keys.
VSETLANG	Specifies the native language ID to be used with an international forms file.
VSHOWFORM	Updates terminal screen, merging the current form, any data in buffer, any key labels, and any message in window.
VUNLOADFORM	Unloads a specified form from terminal local form storage memory.
VWRITEBATCH	Writes data from data buffer to batch file.

HOW INTRINSICS ARE USED

The VPLUS intrinsics control the interface between forms stored in a forms file, a terminal screen, data entered on a form, and an application. Table 6-1. illustrates the relation between the VPLUS intrinsics, terminal, forms file, application, data file or optional batch file, and the buffer areas in memory used by the intrinsics.

The intrinsics use a buffer area in memory for the form definition and another buffer area for the data. A third area in memory is used as a buffer for the window area, the line on the form to which error and other messages are sent. User-defined function key labels also reside in a buffer area while they are active. There is a buffer area for global labels and another for form-specific labels. These buffer areas are managed by VPLUS and are in addition to any buffers the application may define.

Form Definition Area

A form displayed on the terminal screen consists of protected areas (headings, labels, titles, display-only fields) and the unprotected areas (fields) into which data can be entered by a user.

Any form written to the terminal screen by VSHOWFORM already resides as a form image in memory. It may also be stored in terminal local form storage. The form image contains the description of all the protected areas on the form, except for "display only" fields. It also contains the visual enhancements for all fields, protected and unprotected. The form image can be loaded from the forms file into local form storage by VLOADFORMS. Or the form image is read from the forms file into memory by a call to VGETNEXTFORM. Then it is written to the terminal screen or loaded into local form storage depending on options that can be set in VREADFIELDS or VSHOWFORM.

Associated with each form is a set of data specifications — field attributes and processing specifications defined in FORMSPEC. These specifications are read from the forms file by VGETNEXTFORM and reside in memory during execution of the form. The field attributes can be dynamically altered using VCHANGEFIELD.

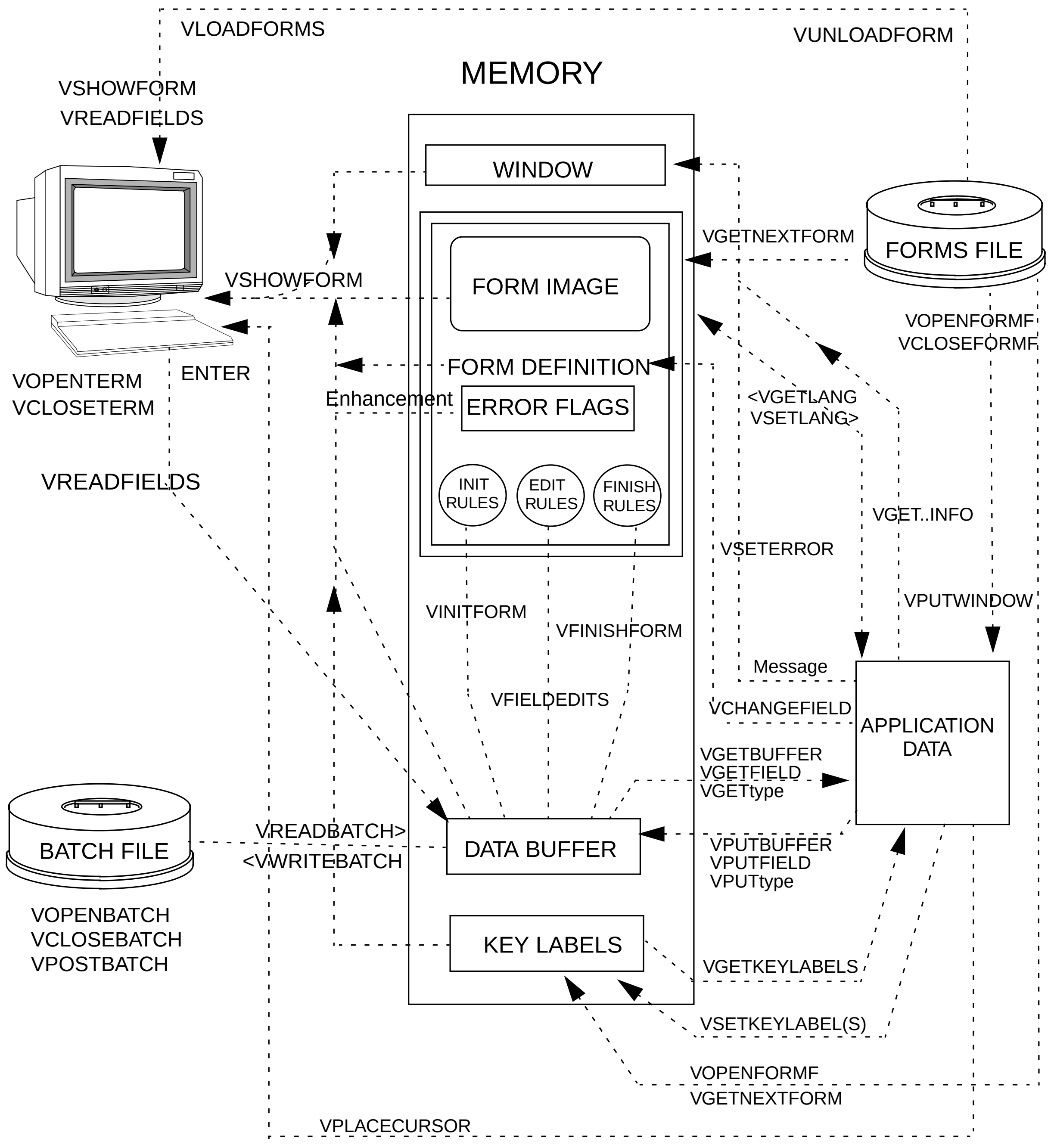
Data Buffer Area

Besides the form definition, there is a data buffer area in memory that contains data for all the fields (unprotected and display only) defined for the form. These fields reside in the buffer in the order they are defined on the screen, from left to right, top to bottom.

When **ENTER** is pressed at the terminal, VREADFIELDS transfers the user-entered data from the screen to the data buffer. Before data is entered, if there are any initializations, VINITFORM sets the appropriate fields in the buffer to initial values. In the field phase, VFIELDEDITS verifies and possibly modifies the user-entered data in the data buffer according to any edit specifications defined for the fields. VFINISHFORM performs any final form modifications specified in the "finish" phase. Data for display-only fields may be written to the buffer by VINITFORM, VFIELDEDITS, or VFINISHFORM. Any changes to the data buffer are displayed at the terminal by the next VSHOWFORM.

An application can send data to the buffer with VPUTBUFFER or to a single field in the buffer with VPUTFIELD or VPUTtype. Conversely, the data buffer can be copied to an application with VGETBUFFER, or a single field can be copied with VGETFIELD or VGETtype.

Figure 6-1. Operation of VPLUS Intrinsics



As is done in ENTRY, during data collection, data in the data buffer can be written to the batch file by VWRITEBATCH. The data in the batch file can be read from the batch file to the data buffer by VREADBATCH. Typically, data is read from the batch file for display at the terminal during browse and modify operations.

Window Area

If the field editing specifications detect an error in data entry or data movement, an error flag is set for the field. When called, VERRMSG returns the message associated with the error, and VPUTWINDOW puts the message into the window. If an error is detected by an application, VSETERROR may be used to set the error flag, and also to put any message associated with the error into the window. Any non-error messages generated by an application are put in the window by VPUTWINDOW. A subsequent call to VSHOWFORM displays the contents of the window at the terminal.

Key Label Area

If there are user-defined function key labels, these labels are transferred to the key label area of memory. There is a key label area for global function key labels and another area for current form labels (global labels are associated with a forms file; form labels are associated only with the current form).

Labels may be defined with FORMSPEC. Alternately, VSETKEYLABEL or VSETKEYLABELS may be used to override existing labels with labels from the application. Global labels defined with FORMSPEC are transferred to the global key label area when the forms file is opened by VOPENFORMF. Form labels defined by FORMSPEC are transferred to the form key label area by VGETNEXTFORM. A global or current form label specified by VSETKEYLABEL is copied into the appropriate key label area from an application. VSETKEYLABELS functions in the same manner for multiple label definitions. Labels are displayed on HP terminals by a call to VSHOWFORM.

CALLING VPLUS INTRINSICS

The examples in Table 6-2. show the format for calls to the VPLUS intrinsics from each language, where:

intrinsicname	Identifies the intrinsic being called.
parameter	At least one parameter is required for each intrinsic; the particular parameters are listed in the formats for the individual intrinsic descriptions.

Table 6-2. Examples of Intrinsic Call Formats for Each Language

Language	**Intrinsic Call Format**
COBOL	CALL *"intrinsicname"* USING *parameter*1 [, *parameter*2]...
FORTRAN	CALL *intrinsicname* (*parameter*1 [,*parameter*2]...)
BASIC	*label* CALL *intrinsicname*(*parameter*1[,*parameter*2]...)
PASCAL	*intrinsicname*(*parameter*1[,*parameter*2]...);
SPL	*intrinsicname*(*parameter*1[,*parameter*2]...);

In order to provide consistency between calls from different programming languages, the following rules apply to all parameters:

- Parameters are passed by reference; this means that a literal value cannot be used as a parameter. The exception is VSETLANG, which has one parameter that is passed by value.
- No condition codes are returned; the status of the call is returned in a status word included as part of the *comarea* parameter specified in every intrinsic call.
- Return type intrinsics are not allowed; any values returned by the intrinsic are sent to the *comarea* or to a passed parameter.

Parameter Types

The data types that are used in VPLUS intrinsics are shown in Table 6-3. Note that not all types are allowed for all languages.

Table 6-3. Data Types Used for Various Languages

Data Type	COBOL	FORTRAN	BASIC	PASCAL	SPL
Character	DISPLAY PIC X(n)	Character	String	Packed Array of Char	Byte Array
Two-byte Integer	COMP PIC S9 thru PIC S9(4)	Integer*2	Integer	Subrange -32768.. 32767	Integer
Unsigned Two-byte Integer	COMP PIC 9 thru PIC 9(4)	Logical	Integer (with value <32767)	Subrange 0..65535	Logical
Four-byte Integer	COMP PIC S9(5) thru PIC S9(9)	Integer*4	Integer Integer[1]	Integer	Double
Real Four-byte	—	Real	Real	Real	Real
Long 8-byte	—	Double Precision	Long	Longreal	Long

[1]In BASIC, a double integer can be represented by two consecutive integers:

- the first contains the high-order digits of values above 32767 *or* is zero,
- the second contains the low-order digits of values above 32767 *or* the entire value up to 32767.

The VPLUS parameters use only data types that are available in all programming languages: character, integer, logical, and double integer. The only exceptions are the transfer and conversion intrinsics, `VPUTtype` and `VGETtype`, which use real and long.

Each parameter is described according to its generic type (character, integer, logical, or double integer). This table is provided for those languages that do not call their data types by these particular names. For example, if you are coding in COBOL and a parameter is specified as logical, you can determine from this table that it is an unsigned computational item that uses from one to four digits.

COMMUNICATION AREA

Every application that calls VPLUS intrinsics must allocate a data area in the application for communication with the intrinsics. This area (called the *comarea*) is the first, and often the only, parameter in every call to a VPLUS intrinsic. Table 6-4. briefly outlines the contents of this communication area; Table 6-5. outlines the items used for data capture devices. It is essential to successful operation of your application that the *comarea* be defined exactly as shown in these tables; items that are "Reserved for system use" should not be set or used after they are initialized. The *comarea* items listed in Table 6-4. and are defined below.

Table 6-4. Outline of Communication Area Contents

Data Type	Position	Offset	Name	Function
Integer (Two-byte)	1	0	*cstatus*	status, error returns
	2	1	*language*	programming language of calling program
	3	2	*comarealen*	length of *comarea* length of *comarea* in two-byte words
	4	3	*usrbuflen*	*comarea* extension length (BASIC)
	5	4	*cmode*	current mode (collect or browse)
	6	5	*lastkey*	code of last key pressed
	7	6	*numerrs*	number of errors in current form
	8	7	*windowenh*	code for window enhancement
	9	8	*multiusage*	Child of previous form
	10	9	*labeloption*	function key label indicator
Character Array (16-byte)	11-18	10-17	*cfname*	current form name (15 characters)
	19-26	18-25	*nfname*	next form name (15 characters)
Integer (Two-byte)	27	26	*repeatapp*	current form flag (repeat/append)
	28	27	*freezapp*	next form flag (freeze/append)
	29	28	*cfnumlines*	number of lines in current form
	30	29	*dbuflen*	data buffer length (in bytes)
Integer (Two-byte)	31	30	Reserved for system use.	—
Logical (Two-byte)	32	31	*lookahead*	flag for background form load in VREADFIELDS
	33	32	*deleteflag*	delete current batch record

Table 6-4. Outline of Communication Area Contents

Data Type	Position	Offset	Name	Function
	34	33	*showcontrol*	control flags for VSHOWFORM
Integer (Two-byte)	35	34	Reserved for system use.	—
Integer (Two-byte)	36	35	*printfilnum*	MPE file number of forms file print file
	37	36	*filerrnum*	MPE file error number from FCHECK
	38	37	*errfilenum*	MPE file number, error message file
	39	38	*formstoresize*	number of forms in buffer
Integer (Two-byte)	40-42	39-41	Reserved for system use.	—
Integer (4-byte)	43-44	42-43	*numrecs*	number of records in batch file
	45-46	44-45	*recnum*	record # of current batch record
Integer (Two-byte)	47-48	46-47	Reserved for system use.	—
Logical (Two-byte)	49	48	*filen*	MPE file number of terminal
Integer (Two-byte)	50-54	49-53	Reserved for system use	—
Logical (Two-byte)	55	54	*retries**	max number of retries
	56	55	*termoptions**	suppress msgs. and autoread
	57	56	*environ*	term environment: Sys LDEV
	58	57	*usertime**	user defined time out period in seconds
	59	58	*identifier*	type of terminal
	60	59	*labinfo*	number of function keys; length
Integer (Two-byte)	61-64	60-63	Reserved for system use	—
	65	64	*buffercontrol*	control ARB processing
	66	65	*bufferstatus*	indicate successful data conversion
	68-70	67-69	Reserved for system use	—
Refer to Table 6-5.	71-85	70-84	—	These *comarea* items are only referenced when using data capture devices.

Not supported on data capture devices.

Table 6-5. Communication Area Contents for Data Capture Devices

Data Type	Position	Offset	Name	Function
Integer (Two-byte)	61-70	60-69	Reserved for system use	—
	71	70	*numflds*	number of fields on current form
	72	71	*splitpause*	length of pause (in seconds)
	73	72	*leftmodule*	type of terminal options
	74	73	*rightmodule*	type of terminal options
	75	74	*keyboard*	type of keyboard
	76	75	*display*	type of display
	77	76	*keyboardover*	whether to override or not
Character (Two-byte)	78	77	*errorlight*	error detection light
Logical (two-byte)	79-80	78-79	*userlightson*	Lights on/off indicator
Integer (two-byte)	81-85	80-84	Reserved for system use	—

NOTE The *comarea* size required for each feature can be summarized as follows:

FEATURE	COMAREALEN
Data Capture Terminal	85
ARB Feature	70
All Others	60

The position of each item in *comarea* is given as a two-byte word offset in the table; for SPL programs the word offset starts with zero; for all other programs, it starts with one.

The *comarea* must be at least 60 two-byte words long (120 bytes) **unless** you are working with an ARB. To take advantage of the ARB feature, the *comarea* must be **at least 70 words long**. The COMAREA array size must be at least as large as the sum of the values of *comarealen* plus *usrbuflen*. For BASIC programs, the area must be extended to include space for the form and data buffers and for internal tables. For non-BASIC programs, the DL area is used for these buffers and internal tables, and the communication area need not be extended. However, non-BASIC programs must be careful not to use the DL area for other purposes when using VPLUS. Refer to Appendix E for more information on the DL area.

Before calling the first VPLUS intrinsic, you should initialize the entire *comarea* to binary zeroes. Do not change *comarea* values between calls except under documented conditions. Then, if you are coding in a programming language other than COBOL, you must set *language* to the code for the programming language you are using. If you are coding in BASIC, you must set *usrbuflen* to the number of two-byte words needed for the comarea extension. This value is displayed on the forms file listing generated by FORMSPEC. Finally, in order to provide for possible future extensions to *comarea*, set *comarealen* to its current total length.

Parameters

cstatus Two-byte integer to which the intrinsic status is returned. Set to zero if the call is successful; to a nonzero value if an error occurs. If the error is an MPE file error, a file error number is also returned to *filerrnum* (Refer to Appendix B for a list of the error codes that may be returned to *cstatus* with their meaning.) It is up to the user to provide error-handling routines and to reset *cstatus*.

language Two-byte integer that indicates the programming language of the calling program:

- 0 = COBOL II/III
- 1 = BASIC
- 2 = FORTRAN 66
- 3 = SPL
- 5 = PASCAL//HP FORTRAN 77//HP BUSINESS BASIC //C/XL

The *language* must be set by the calling program before any intrinsic is called.

comarealen Two-byte integer that indicates the total length of *comarea* (use 6 0 for terminals; 85 for data capture devices). The *comarealen* should be specified for each *comarea* to simplify future changes to the length of *comarea*.

usrbuflen For BASIC programs only, *usrbuflen* must be set to the number of two-byte words to be appended to *comarea* for the form and data buffers and internal tables. (Note that the number of bytes required for this extension is printed when a form is listed through FORMSPEC.) The *usrbuflen* need not be set by other languages since VPLUS automatically uses the DL area for buffers and internal tables. The *usrbuflen* does not include the *comarea* length specified in *comarealen*, but it must be contiguous to and immediately follow *comarea*.

cmode Indicates whether current mode of data entry is collect.

- 0 = Collect Mode
- 1 = Browse Mode

If you want to save the form sequence for $RETURN, you must set *cmode* to zero before calling VGETNEXTFORM.

lastkey Two-byte integer set to a number between -256 and 26 (or -999) by VREADFIELDS to indicate user response, which can be **ENTER** or the last function key pressed at a standard terminal, the last field touched for a touch terminal, or the last key pressed for data capture devices. The value returned can be interpreted as follows:

For standard terminals, 0-8 is returned:

- `0` = **ENTER**
- `1 -8` = f1-f8

For touch, in addition to 0-8, -256 to -1 (or -999) is returned:

- `-1 to -256` = field number of touched field
- `-999` = no field in area touched by user

For data capture devices, - 1 to 26 is returned:

- `-1` = Attention key
- `0` = enter
- `1-26` = A-Z

`numerrs`	Two-byte integer set to the number of errors found when a form is edited according to FORMSPEC edit specifications.
`windowenh`	Two-byte integer in which the right byte contains a USASCII code for the window line enhancement. The code specifies any combination of enhancements according to Table 6-6.; zero indicates no enhancement. For example, to indicate Half -Bright, Inverse Video, *`windowenh`* is set to the character J.

Specifying an enhancement code with *`windowenh`* is independent, and in addition to, any color enhancement specified with FORMSPEC.

On data capture devices any enhancement except blinking is ignored. The entire display blinks, not just the field in error.

Table 6-6. Codes for the Window Line Enhancement

	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
Half-Bright									X	X	X	X	X	X	X	X
Underline					X	X	X	X					X	X	X	X
Inverse Video			X	X			X	X			X	X			X	X
Blinking		X		X		X		X		X		X		X		X
Stop enhancement	X															

multiusage Two-byte integer that indicates whether the current form is in the same family as the previous form but is not the parent form.

- 0 = Not in the current family
- 1 = Child or sibling to previous form

labeloption Two-byte integer that indicates whether function key labels are to be used.

- 0 = Default function key labels are used (F1 - F8).
- 1 = User-defined labels are to be used

Space to store the labels defined either in FORMSPEC or specified with VSETKEYLABELS is allocated in the user stack by VOPENFORMF.

The *labeloption* must be set prior to a call to VOPENFORMF so that it can be determined whether or not to allocate space for user-defined function key labels.

cfname A 16-byte character array containing the name of the current form (15 characters). Updated by VGETNEXTFORM.

NOTE All *comarea* entries start on two- or four-byte word boundaries. Thus, the 16-byte character array items *cfname* and *nfname* are each followed by one filler byte that is not part of the name.

nfname A 16-byte character array containing the name of the next form (15 characters). Required by VGETNEXTFORM and updated when necessary by VGETNEXTFORM. The *nfname* may be set by an application. The *nfname* may contain one of the following values:

formname	Identifies the form to be read from the forms file.
$REFRESH	Clear and reset the terminal. Set the internal flags to indicate to VSHOWFORM to redisplay the current form, window and data buffer. For terminals with local form storage, clear form storage memory and reinitialize the form storage directory.
$RETURN	Retrieve previous different form; if current form is the

head form, the current form is retrieved. There is a maximum of eight forms; after eight, each additional form replaces the most recent form.

`$HEAD`	Retrieve first form displayed when the forms file is executed (the "head" form).
`$END`	For ENTRY only, terminate execution of VGETNEXTFORM; return to calling program without resetting any *comarea* items.

repeatapp Two-byte integer that indicates whether the current form is a repeating form and, if so, whether it is to be appended to itself:

- `0` = Normal sequence, neither repeat nor append
- `1` = Repeat current form
- `2` = Repeat current form, appending it to itself

freezapp Two-byte integer that indicates whether the screen is to be cleared when the next form is displayed, or whether next form is to be appended to current form, and, if appended, whether current form is to be frozen on the screen. If the current form is frozen, it remains on the screen and the next form is rolled off when the screen is full; otherwise, the current form is rolled off the screen when the screen is full. The *freezeapp* is specified as:

- `0` = Clear screen, neither freeze nor append current form
- `1` = Append next form to current form
- `2` = Freeze current form and append next form to it

cfnumlines Two-byte integer that specifies the number of lines in the current form. The *cfnumlines* is used by VPRINTFORM to print the form off-line.

dbuflen Two-byte integer that specifies the number of bytes in the data buffer for the current form. Set by VGETNEXTFORM, this length is the sum of all the concatenated data fields in the form, including any display-only fields.

lookahead Two-byte integer that indicates whether or not look-ahead form loading is to occur when VREADFIELDS is called.

`0 - ON`	The next form, as defined in FORMSPEC, is loaded before the current form is read if point-to-point data communications is being used. If multipoint data communications is being used, the next form is loaded after the current form is read. The least recently used form in the terminal's local form storage memory could be purged to make room for the new form if there is no room in the terminal or there are no available entries in the form storage directory.
`1 - OFF`	No look-ahead form loading is performed, and no forms in the terminal's local form storage memory are purged (however, forms could still be purged if VSHOWFORM is called to preload forms; see the discussion of VSHOWFORM later in

this section).

If *lookahead* is zero and a family member of the form specified as the next form is already in the terminal's local form storage memory, the new form is not loaded. Instead, the family member is changed into the required form when it is displayed. The *lookahead* feature is ignored if the terminal does not have form storage capability.

deleteflag Two-byte logical flag that indicates whether the current batch record has been or is to be deleted. The *deleteflag* is returned by VREADBATCH and used by VWRITEBATCH:

- `FALSE (all zeros)` = Current batch record not deleted
- `TRUE (all ones)` = Delete current batch record

showcontrol Control options for VSHOWFORM or VREADFIELDS. These bits remain set until an application resets them. The following bits can be set:

`bit 15 = 1`	Force form to be written to the terminal screen; overrides VPLUS optimization.
`14 = 1`	Force data and field enhancements to be written to the terminal screen; overrides VPLUS optimization.
`13 = 1`	Force window line to be written to the terminal screen; overrides VPLUS optimization.
`12 = 0`	Stop after displaying a form without fields (default); data capture devices only.
`1`	Do not stop after displaying a form without fields; data capture devices only.
`11 = 0`	Do not put a right closing bracket (]) on all input fields (default); data capture devices only.
`1`	Put a right closing bracket (]) on all input fields; data capture devices only.
`10 = 0`	Enable the keyboard for input (default).
`1`	Do not enable the keyboard.
`9 = 0`	Do not preload form into the terminal's local form storage memory but write it directly to the screen (however, VSHOWFORM always checks to see if the form is in local storage already) (default).
`1`	Preload form into the terminal's local form storage memory if it is not already there. Forms could be purged from local form storage if there is not room for the form (this could occur even when *lookahead* is 1).
`8 = 0`	Do not enable **AIDS**, **MODES**, or **USERKEYS** (default).
`1`	Enable **AIDS**, **MODES**, and **USERKEYS**. These keys are enabled after a form is displayed on the screen and remain

enabled until bit 8 is reset to 0.

`7-1 = 0`	The undefined bits (7-1) must be initialized to zero.
`0 = 0`	Do not enable the touch feature (default).
`1`	Enable the touch feature.

Bits 15, 14, and 13 are used to override VSHOWFORM optimization (where only information that has changed since the last call to VSHOWFORM is written to the terminal screen). Bits 12 and 11 are used with VREADFIELDS on data capture devices. Bit 10 is used when consecutive calls are made to VSHOWFORM (no call to VREADFIELDS in between). Bit 9 is used with terminals having local form storage capability.

Bit 8 is used to allow access to the function control keys on the HP 239X and HP 262X terminals. For example, device control and device mode keys can be used for local printing to an integral printer. Enabling the function control keys using bit 8 is recommended for experienced users only. Misuse or accidental use of certain function control keys could yield undesirable results.

Bit 0 is used to activate touch available with terminals with the touch feature. VREADFIELDS returns *lastkey* with the field number of the touched field (as a negative number).

printfilnum MPE file number to which a form is printed by VPRINTFORM. The formal designator is FORMLIST.

filterrnum MPE file error number (FCHECK number) returned by VPLUS intrinsics when an MPE file error occurs. (Refer to Appendix B for a list of the errors that return a number to *filerrnum*).

errfilenum MPE file number of the error message file used by VERRMSG.

formstoresize Two-byte integer that indicates the number of entries allowed in the form storage directory.

`-1`	No local form storage is to be performed. For the HP 2626A terminal the workspace configuration is under user control. VPLUS does not modify terminal workspace, window, or datacomm configuration, and either terminal port can be used.
`0`	No local form storage is to be performed. For the HP 2626A terminal VPLUS reconfigures the workspaces and windows to the terminal's default values, but only the first terminal port can be used.
`1..255`	One through 255 forms can be stored locally. For the HP 2626A terminal any value greater than four defaults to four (allowing an application to run on either the HP 2626A, HP 2394A, or HP 2624B terminals), where four indicates that one form is the displayed form and three forms are stored in the HP 2626A local forms storage. VPLUS reconfigures the HP 2626A terminal workspaces

and windows to support local form storage, but only the first terminal port can be used.

The *formstoresize* allows the user to control the number of entries allowed in the form storage directory on the user stack and must be set prior to opening the terminal and the forms file. The space required is listed on the forms file listing generated by FORMSPEC.

When designing forms for local form storage terminals, keep in mind that large forms need a substantial amount of terminal display memory. For forms that are greater than 24 lines, so much terminal display memory may be used that insufficient terminal memory is available for form storage. If so, VPLUS resets *formstoresize* to zero and no local form storage is performed.

numrecs Four-byte integer that contains the number of nondeleted records in the current batch file. It is used by the V..BATCH intrinsics.

recnum Four-byte integer set to the current record number in the batch file. (Note that record numbers start with zero.) The *recnum* must be set by the program before writing to or reading from the batch file. It is used by the VREADBATCH and VWRITEBATCH intrinsics.

filen MPE file number used to identify the terminal.

retries Maximum number of retries.

- value = 0 Use default value (four retries)
- value > 0 Use this value as maximum
- value < 0 do not perform any retries

termoptions Terminal control options:

bits

0-8	reserved for system use
9-10	01 = enables **ENTER** /function key timeout in VREADFIELDS. 11 or 00 = disables **ENTER** /function key timeout in VREADFIELDS. (default)
11-12	10 = do not clear the screen at terminal open or close.
13-14	01 = enables AUTOREAD in VREADFIELDS. 11 or 00 = disables AUTOREAD. (default) (AUTOREAD causes the terminal to do a programmatic enter instead of waiting for the user to press **ENTER**.)
15	0 = Sound bell as usual, (default) 1 = Suppress bell.

NOTE In programming languages which do not have the capability of setting single bits, programmers should add or subtract appropriate amounts to change single bits. For example, if *termoptions* was initialized entirely to zeros and the programmer wishes to enable the AUTOREAD, the following statement should be issued in COBOL:

```
ADD 2 TO TERM-OPTIONS.
```

Conversely, if disabling the AUTOREAD feature, the programmer should subtract 2 from *termoptions*. For more information on the AUTOREAD feature, please consult Appendix G.

environ	First byte is the logical device number of the terminal. The remaining byte is reserved for system use.
usertime	If enabled, the value in this position is used as the number of seconds to wait for either **ENTER** or a function key to be pressed. Consult Appendix G for more information regarding enabling user timeouts.
identifier	VPLUS identifier for the terminal model being used. See Appendix G for the interpretation of the contents of this word.
labinfo	First byte is the length of key labels (in bytes); Second byte is the number of key labels that VPLUS uses on the terminal.
buffercontrol	Controls the ARB options. The bit settings are as follows:
	Bit *15 (ARB option)* 0 = return data "raw": no reordering or conversion.
	1 = reorder and convert data according to ARB specifications.
	The remaining bits must be set to zero. Note that VGETBUFFER will not change these settings (see Section 6, VGETBUFFER, for special considerations).
bufferstatus	Return parameter indicating that VGET/PUTBUFFER performed the data conversion successfully (but 15 in offset 65 will be set to 1).

NOTE STATUS in the *COMAREA* will be non-zero if the conversion is not successful.

In addition to the above *comarea* descriptions, the data capture devices use the following items:

Parameters

splitpause Length of time in seconds to pause between the presentation of lines of text on the single line alpha display. Default is 3 seconds.

- `-1` - wait for user to press key
- `0` - do not pause
- `>0` - pause specified number of seconds

leftmodule MPE determines which, if any, module is present.

- `0` - no module
- `1` - printer

- `2` - multifunction reader
- `3` - RS232 interface
- `4` - typeV badge reader
- `5` - magstripe reader
- `6` - bar code reader
- `7` - HP-IB interface

`rightmodule` MPE determines which, if any, module is present.

- `0` - no module
- `1` - printer
- `2` - multifunction reader
- `3` - RS232 interface
- `4` - typeV badge reader
- `5` - magstripe reader
- `6` - bar code reader
- `7` - HP-IB interface

NOTE VPLUS does not communicate with the RS-232 interface (3) or the HP-IB interface (7).

`keyboard` Type of keyboard used with terminal.

- `0` - HP 3077-no keyboard.
- `1` - standard keyboard (12 function keys with values of -1,0,17...26).
- `2` - alphanumeric keyboard (28 function keys with values of - 1,0...26).

`display` Indicates the terminal type of display.

- `0` - numeric display
- `1` - alphanumeric display
- `2` - mini-CRT display

VPLUS does not support the terminal that has numeric display.

`keyboardover` Two-byte integer determining whether to override input on the keyboard or not. Default is 0.

- `-1` - Override and enable the keyboard without regard to forms design.
- `0` - Do not override. Allow input from the devices specified during the form design (in the Processing Specification area).

`errorlight` Indicates which light to be turned on when an error is detected. The default is `E`, but this can be changed to @ or any letter from `A` to `P`. The second byte is reserved for system use.

lightsOn Two two-byte integers to indicate whether or not lights are to be turned on during run time. The turning on of lights here does not affect the lights turned on during form design (in the Process Specification area of the form).

- 0 - OFF
- 1 - ON

The default value is OFF, but this may be changed as follows:

```
TWO-BYTE WORD 1, a "1" in:
bit position 0 > turns light of key "@" ON
              1                        "A" ON
              2                        "B" ON
             ...                       ...
             14                        "N" ON
             15                        "O" ON
 TWO-BYTE WORD 2, a "1" in:
 bit position 0 > turns light of key "P" ON
```

The remaining bits are reserved for system use.

ERROR HANDLING

There are basically two types of errors that can occur as a result of calling VPLUS intrinsics. The first type consists of errors in the intrinsic call itself or in an attempt to access a file used by the called intrinsic. The second type are errors detected by editing data entered into FORMSPEC forms. These two error types are handled differently by VPLUS.

Intrinsic Call or File Errors

If a call to a VPLUS intrinsic causes an error so that the intrinsic cannot be executed correctly, or if an MPE file error occurs as a result of an attempt to access a file with a VPLUS intrinsic, the *comarea* word *cstatus* is set to a nonzero value. In addition, the number associated with an MPE file error is stored in the *comarea* word, *filerrnum*.

When *cstatus* is not zero, any subsequent VPLUS intrinsics called by the application return to the application without executing. As part of good programming practice, check *cstatus* after each call, report the error, and then reset *cstatus*. There is one exception — VERRMSG uses *cstatus* to determine the error number. If *cstatus* is set to zero (indicating no error) prior to the call to VERRMSG, then no message is returned.

Editing Errors

Field processing may be specified in the forms description with FORMSPEC and checked by VFIELDEDITS, VINITFORM, or VFINISHFORM; or editing may be provided by user routines in an application.

Each field in a FORMSPEC form has an error flag associated with it. When one of the VPLUS intrinsics that performs field processing (VFIELDEDITS, VINITFORM or VFINISHFORM) detects a field error, it sets the error flag for that field. It also increments *numerrs*, the word in *comarea* that contains the total number of fields with errors in each form. If a user-provided editing routine detects a field error, the program must call the VPLUS intrinsic VSETERROR in order to set the field error flag and increment *numerrs*. The *cstatus* item is not set when an editing error is detected and subsequent intrinsics may be executed without resetting this *comarea* item.

If new data is written to a field in the data buffer that had an error, the error flag for the field is cleared and *nurnerrs* is decremented. VPUTBUFFER, VPUTFIELD, or VPUTtype are the intrinsics that can correct field errors and decrement *numerrs*. The intrinsic VREADFIELDS resets *numerrs* to zero when it reads new data into the buffer from the terminal. If *numerrs* has been set to a nonzero value by one of the VPLUS edit intrinsics, then VREADFIELDS with reset it to zero. Also, when VGETNEXTFORM is called, *nurnerrs* is reset to zero.

Error Messages

Messages associated with all VPLUS-detected errors can be retrieved by a call to VERRMSG. This intrinsic uses the error message file whose MPE file number is kept in the *comarea* word, *errfilenum* The error message file contains internal error numbers linked to particular field errors. Error messages may be general VPLUS messages (see Appendix B), or custom messages specified during forms design with FORMSPEC. In either case, the message is returned to the calling program by VERRMSG. VERRMSG determines the type of the error by examining *cstatus* and *numerrs*. If *cstatus* is not zero, its value indicates a particular intrinsic call error. If *cstatus* is zero and *numerrs* is set, VERRMSG knows the error is an editing error and uses internal values to locate the error in the error message file. If an editing error is detected by a user routine, the program must provide its own message when it calls VSETERROR to set the error flag for the field.

Determining Fields in Error

VGETFORMINFO provides a method of determining which fields are flagged in error by the editing routines. VFIELDEDITS sets an error flag for each field that failed edit checks. Only the error number of the first field in error and the number of errors have been returned to an application by VFIELDEDITS via the *comarea*. All fields in error are enhanced with the error enhancement if VSHOWFORM is called after VFIELDEDITS. To enable applications to determine which fields are in error for a given form at run-time, VGETFORMINFO optionally returns data about these error flags to show which fields failed the edit checks.

USING TERMINAL FEATURES

VPLUS provides a way to take advantage of special terminal features, using processing specifications (refer to Section 4), VPLUS intrinsics, and selections on the FORMSPEC menus (refer to Section 3). Two of the special terminal features that can be controlled with VPLUS intrinsics are touch and local form storage, as described below. Refer to Appendix G for a list of terminals that support these features.

The Touch Feature

VPLUS supports the touch feature available with specified terminals, as listed in Appendix G. The touch feature is aimed to improve programmer productivity for the design of intuitive, friendly user interfaces.

To activate the touch feature, an application must set bit 0 of the *showcontrol* word in the comarea to 1 before calling VSHOWFORM. This feature will remain activated until the *showcontrol* bit 0 is set to 0, or until VCLOSETERM is called.

With the touch enhancement enabled, VREADFIELDS returns a number when a field on the form is touched. This number is the negative of the field number that was assigned by FORMSPEC at form design time. All fields as currently defined on a form return a negative of their assigned field number when touched. No additional definition of touch fields is necessary. The number returned is negative in order to distinguish it from the positive numbers returned for function keys. In this manner, fields on the form can be treated just like function keys.

Thus, touch applications can be designed using the same VPLUS intrinsics and the same forms file. The negative field number returned to the *lastkey* field in the *comarea* can then be interpreted for further processing.

Local Form Storage

Certain terminals allow forms to be stored locally in terminal memory, which can reduce the overhead of writing a form image from the form definition area of memory. Frequently displayed forms can be loaded into the terminal to be written directly to the screen. Only form images are loaded — not the associated data. The HP 2626A terminal can store as many as four forms locally. The HP 2624B and HP 2394A terminal can store a maximum of 255 forms locally depending on the size of the forms and the size of terminal memory available.

The VPLUS intrinsics and communication area items activate and control local form storage. The same intrinsics and variables are used with local form storage terminals, but there are some differences as to how the form storage is handled. These differences, along with general information about using form storage, are described here. The specifics of the intrinsics involved are contained in the individual intrinsic descriptions later in this section.

The HP 2626A vs. the HP 2624B/HP 2394A

VPLUS utilizes the workspace/window feature of the HP 2626A terminal, configuring terminal memory into from one to four workspaces to store forms. Total available space is equal to 119 lines of forms, with the minimum size of a workspace being 26 lines. One full-screen window is used to display a form.

A form is displayed by attaching a workspace to the window. Any enhancements and data entered onto the form are part of the form as long as it is in the terminal — a "fresh" form is not available in local storage. If a fresh copy of the form is to be displayed, it must be written from the form definition area of system memory.

One workspace is always attached to the window and displayed, even if there are no forms loaded. The workspace being displayed is frozen — no form can be loaded into this frozen workspace. Therefore, only three forms can actually be loaded in a single call to VLOADFORMS. To get a fourth form into the terminal, display it in the window with VSHOWFORM.

Forms can be purged from the terminal if there is not room for the form you are trying to load. Forms are purged on a "least recently used" basis. During a load operation, each form is frozen until all forms have been loaded, thus preventing loading and purging in a single operation.

If you want to purge the currently displayed form from the terminal, it is not erased from the screen, but the form storage directory entry for this workspace is marked available. (The form storage directory is discussed later in this section.)

On the HP 2624B/HP 2394A terminals, forms stored locally are copied from terminal memory to the screen. A fresh copy of the form remains available in local storage. When the terminal is opened, enough memory is reserved to display the largest form in the forms file and this affects how many forms can actually be loaded.

Form Storage Directory

The *comarea* variable *formstoresize*, which must be set before opening the terminal and the forms file, indicates the maximum number of forms you are going to store locally. A form storage directory with an entry for each form is created on the user stack. This directory is used to keep track of loaded forms and of how much terminal memory each form is using. Note that setting this variable does not load any forms — it just reserves space for them. VSHOWFORM searches the directory to determine if a form to be displayed is in local storage.

When a form is purged from local storage, the corresponding directory entry is marked available to indicate that the form is no longer in the terminal and is not taking up any terminal memory space.

NOTE VCLOSETERM deallocates local form storage space — both VOPENTERM and VOPENFORMF must be called if local form storage space is to be reallocated.

Loading Forms

Forms can be loaded into local form storage in the following ways:

- Preload the current form when VSHOWFORM is called by setting *showcontrol* bit 9 to one.
- Load the next form when VREADFIELDS is called by setting *lookahead* to zero.

- Call VLOADFORMS.

When VSHOWFORM is called to display a form, it searches the form storage directory to determine whether or not the form is already in the terminal.

If the form is in local storage, VSHOWFORM displays it on the screen. If the form is not in local storage, VSHOWFORM preloads it into the terminal before displaying it when bit 9 of the *comarea* variable *showcontrol* is set to one.

If there is not enough room in terminal memory for the form, the least recently used form (or forms) is purged from the terminal. If the form is not already in the terminal and is not preloaded into local form storage (bit 9 is zero), it is written to the screen from the form definition area of system memory.

(On the HP 2626A terminal, the form is written from memory when *showcontrol* bit 15 is set to force a write, no matter what the setting of *showcontrol* bit 9 or whether the form is already in the terminal.)

Look-ahead loading can be performed by setting the *comarea* variable *lookahead* to zero before the call to VREADFIELDS. The next form, named by *nfname* in VGETNEXTFORM, is loaded before or after the current form is read depending on the type of data communications being used. If point-to-point is being used, the next form is loaded before the current form is read, If multipoint is being used, the next form is loaded after the current form is read. The least recently used form (or forms) is purged from the terminal if there is not enough room to load the next form.

VREADFIELDS does not check whether or not the next form loads successfully. If the form does not load, the next call to VSHOWFORM retrieves it from the form definition area of system memory.

Forms can also be loaded into local form storage by a call to VLOADFORMS, which simply loads the forms named in the *forms* parameter. A form loaded by this intrinsic is thus already in the terminal when the call to VSHOWFORM is made (recall that VSHOWFORM always checks whether the form is in local storage). Forms are loaded in the order specified in the *forms* parameter as long as there is space in the terminal. Any remaining forms are ignored.

VLOADFORMS checks to see whether or not a form is loaded. During this check, the keyboard is locked briefly. To avoid possible loss of keystrokes and/or entered data do not call VLOADFORMS between VSHOWFORM and VREADFIELDS.

Form Families and Local Storage

VLOADFORMS loads forms strictly by form name so that multiple family members can be loaded at the same time. For look-ahead loading or preloading, to optimize performance, the form is not loaded if a related form is currently being displayed or is already in the terminal. The family member that is in local storage is changed into the required form and displayed by VSHOWFORM (unless *showcontrol* bit 15 is set to force a fresh copy of the form to be written).

Note that form family optimization displays any data associated with the required form. When optimization occurs on the HP 2626A terminal, the form storage directory is updated to reflect the name of the required form rather than the name of the family member that was used to make the change.

Appending Forms and Local Storage

The implementation of appending forms differs between the HP 2626A and the HP 2624B/HP 2394A terminals. On the HP 2626A, the appending form is always written to the screen from the form definition area of memory rather than being displayed from local storage. If, however, the appending form has 24 or more lines (23 or more if a message window is defined), it is handled as an independent form rather than as an appending one. In this case, it can be displayed from local storage.

The workspace for the current form (the one being appended to) is marked available, so the current form cannot be displayed again from local storage.

Form family optimization occurs when the current form is an appended form and is related to the next form to be displayed. The current appended form is changed into the required form when it is displayed.

On the HP 2624B/HP 2394A terminals, the appending form can be copied from local storage.

Purging Forms From Local Storage

The VUNLOADFORM intrinsic is used to purge an unneeded form from local storage to make room for a new form. This intrinsic checks to see whether or not a form is purged. During this cheek, the keyboard is locked briefly. To avoid possible loss of keystrokes and/or entered data, do not call VUNLOADFORM between VSHOWFORM and VREADFIELDS. In addition, the intrinsics VCLOSEFORMF and VCLOSETERM also clear all forms from local storage.

With look-ahead loading, forms can be automatically purged from the terminal to make room for a new form. To protect forms from being purged from local storage, set *lookahead* to one before any other calls to VREADFIELDS or to VLOADFORMS. If *lookahead* is zero, the least recently used forms are purged if necessary. However, when *showcontrol* bit 9 is set to one, forms can be purged to make room even if look-ahead is not enabled.

INTRINSIC DESCRIPTIONS

The intrinsics on the following pages are described in alphabetic order for easy reference. However, this is not the order in which they are normally used. Table 6-7. is provided to group the intrinsics according to the functions they perform. Note that terminal access is performed by only seven intrinsics: VOPENTERM, VCLOSETERM, VPLACECURSOR, VSHOWFORM, VREADFIELDS, VLOADFORMS, and VUNLOADFORM. The remaining intrinsics interact with the form definition, data buffer, window, and key label areas of memory. An example of the order in which the intrinsics are used is shown in Table 6-7.

Table 6-7. Intrinsics by Function Group

Function	**Intrinsic**	
Access to Terminal	VOPENTERM VCLOSETERM VSHOWFORM VPLACECURSOR	VREADFIELDS VLOADFORMS VUNLOADFORM
Access to Forms Definition	VOPENFORMF VGETFIELDINFO VGETFORMINFO VPRINTFORM	VCLOSEFORMF VGETFILEINFO VGETNEXTFORM VCHANGEFIELD
Data Processing	VINITFORM VFINISHFORM	VFIELDEDIT
Data Entry	VCLOSEBATCH VPOSTBATCH VWRITEBATCH	VOPENBATCH VREADBATCH
Programmatic Access to Data	VGETBUFFER VPUTBUFFER VGETtype	VGETFIELD VPUTFIELD VPUTtype
Access to Error/Status Window, Function Key Labels, Native Language ID	VERRMSG VGETKEYLABELS VSETKEYLABELS VGETLANG	VSETERROR VPUTWINDOW VSETKEYLABEL VSETLANG

All intrinsics require that three items in *comarea* (*cstatus, language* and *comarealen*) are set before the intrinsic is called. However, two of these items, *language* and *comarealen*, only need to be initially set, prior to the *first* intrinsic call, since they are not modified by any subsequent intrinsics. In contrast, *cstatus* should be reset before calling any intrinsic after an error occurs (except for the intrinsic VERRMSG that uses *cstatus* in retrieving the associated error message).

No running examples are provided with the intrinsic descriptions. Instead, Appendix A contains sample programs that use many of the intrinsics described in this section.

Figure 6-2. Intrinsic Flow

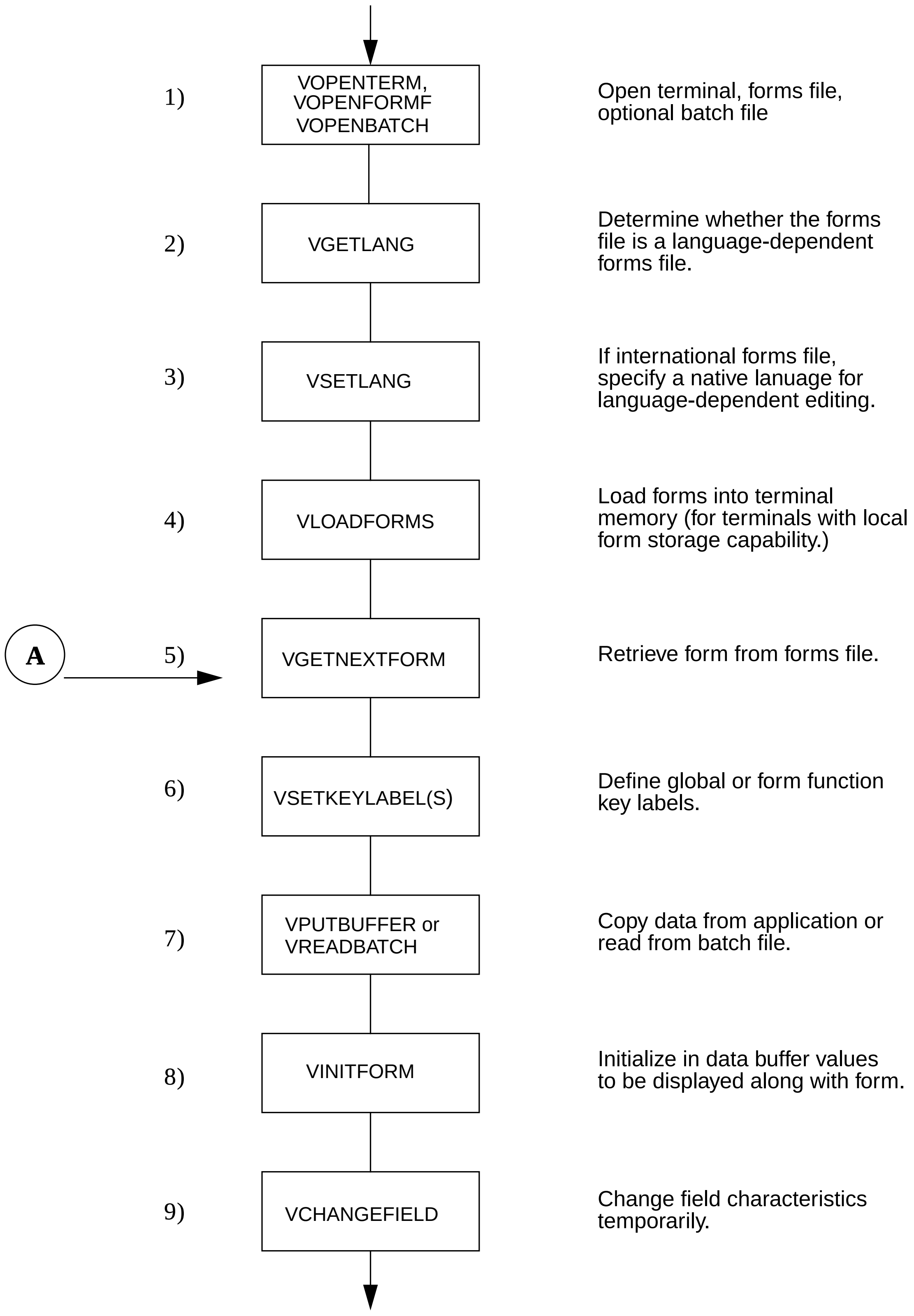
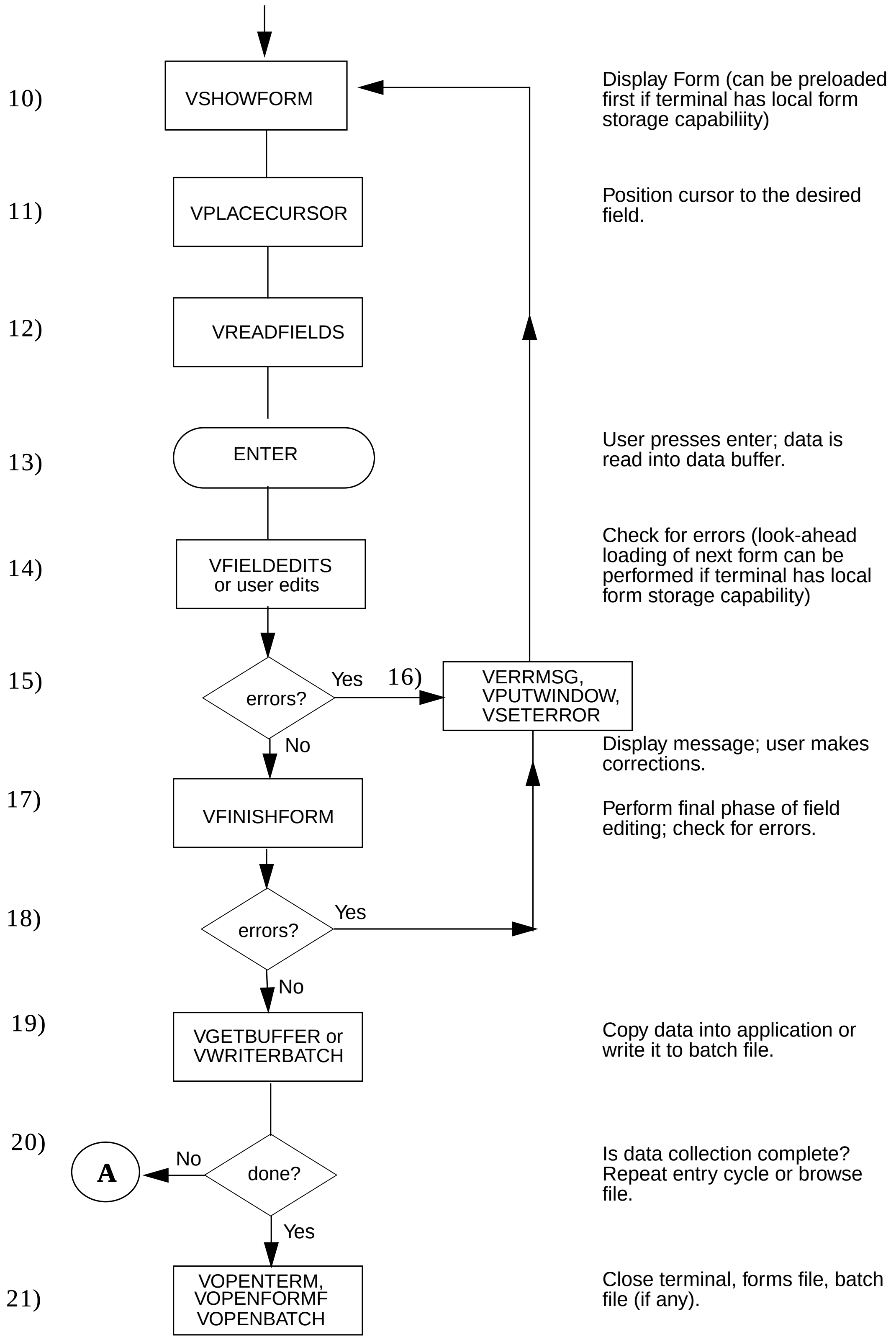

1)
VOPENTERM, VOPENFORMF VOPENBATCH
Open terminal, forms file, optional batch file
2)
VGETLANG
Determine whether the forms file is a language-dependent forms file.
3)
VSETLANG
If international forms file, specify a native lanuage for language-dependent editing.
4)
VLOADFORMS
Load forms into terminal memory (for terminals with local form storage capability.)
A
5)
VGETNEXTFORM
Retrieve form from forms file.
6)
VSETKEYLABEL(S)
Define global or form function key labels.
7)
VPUTBUFFER or VREADBATCH
Copy data from application or read from batch file.
8)
VINITFORM
Initialize in data buffer values to be displayed along with form.
9)
VCHANGEFIELD
Change field characteristics temporarily.

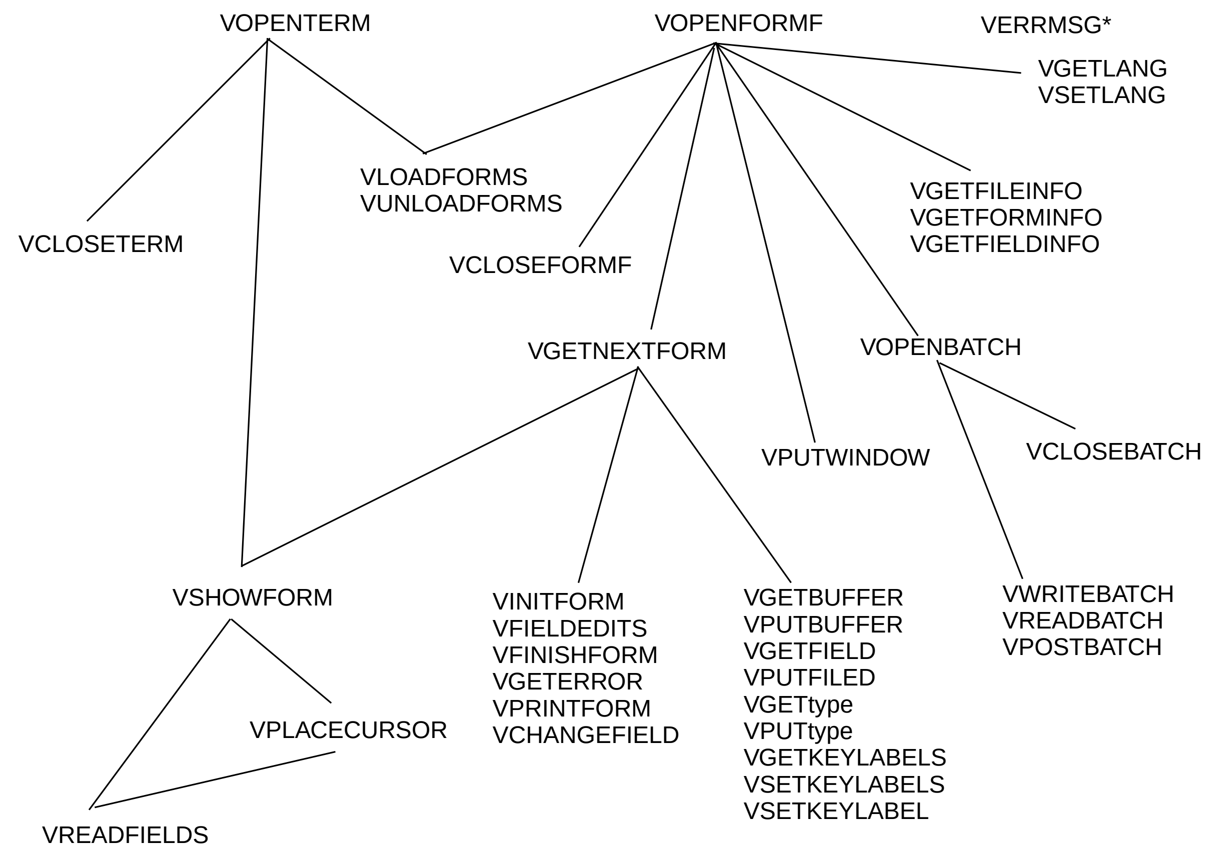
Figure 6-2 Intrinsic Flow Continued:


10)
VSHOWFORM
Display Form (can be preloaded first if terminal has local form storage capabiliity)
11)
VPLACECURSOR
Position cursor to the desired field.
12)
VREADFIELDS
13)
ENTER
User presses enter; data is read into data buffer.
14)
VFIELDEDITS or user edits
Check for errors (look-ahead loading of next form can be performed if terminal has local form storage capability)
15)
errors?
Yes
16)
VERRMSG, VPUTWINDOW, VSETERROR
No
Display message; user makes corrections.
17)
VFINISHFORM
Perform final phase of field editing; check for errors.
18)
errors?
Yes
No
19)
VGETBUFFER or VWRITERBATCH
Copy data into application or write it to batch file.
20)
A
No
done?
Is data collection complete? Repeat entry cycle or browse file.
Yes
21)
VOPENTERM, VOPENFORMF VOPENBATCH
Close terminal, forms file, batch file (if any).

Dependency Between Intrinsics

Certain intrinsics must be called before other intrinsics can be executed. Figure 6-3. illustrates the standard dependencies among intrinsics. For example, the terminal file must be opened before any other intrinsics are called if prompts are to be sent to the terminal. Each of the three files, terminal, forms and batch must be opened before operations can be performed on these files and before the files can be closed.

Figure 6-3. Intrinsics Dependencies



*VERRMSG does not require any previous intrinsics calls.

Note: VGetNextForm must be called before either VShowForm or VReadFields can be done.

VARMSCP

Arms or disarms cursor sensing capability.

Syntax

VARMSCP {comarea,scpenable}

Parameters

comarea	Must be *comarea* name specified when the forms file was opened with VOPENFORMF. If not already set, the following *comarea* items must be set before calling VARMSCP:	
	cstatus	Set to zero.
	comarealen	Set to total number of two-byte words *comarea*. Must be at least 70 words in length.
VARMSCP	May set the following *comarea* item, *cstatus*.	
	cstatus	Set to nonzero value if call is unsuccessful.
scpenable	Two-byte logical variable which determines whether the cursor position sensing is enabled or not.	
	• 0 - Disarm cursor sensing.	
	• 1 - Arm cursor sensing.	

Discussion

When cursor sensing is armed, the data returned after a read, as in VREADFIELDS, is prefixed with an escape sequence which contains the position of the cursor on the screen when the read terminated. This information is used by VGETSCPFIELD and VGETSCPDATA to retrieve the cursor position.

When cursor sensing is disarmed, no cursor position information is available following a read. VREADFIELDS automatically disarms cursor sensing. VARMSCP should be called *prior* to a cursor sensing transaction to arm the cursor sensing function for the next read. This deletes any existing cursor location information.

Example

```
COBOL
CALL “VARMSCP” USING COMAREA SCP-ENABLE.
```

```
SPL
VARMSCP(COMAREA,SCP’ENABLE);
```

These calls arm/disarm cursor sensing depending on whether the second parameter contains a 1 or a 0 respectively.

VBLOCKREAD

The VBLOCKREAD intrinsic reads a block of characters from a terminal in block mode. The syntax and parameter descriptions for this intrinsic are provided below.

```
VBLOCKREAD {COMAREA,BUF,LEN,ACTLEN,TMODE,LOC,BC,TC}
```

COMAREA	The following COMAREA fields must be set before calling VBLOCKREAD, if not already set:

LANGUAGE	Set to code identifying the programming language of the calling program.
COMAREALEN	Set to total number of 2-byte words in COMAREA.

VBLOCKREAD may set the following COMAREA fields:

CSTATUS	Set to nonzero value if call is unsuccessful.
FILERRNUM	Set to file error code if MPE file error.

BUF	Byte array to receive data from the terminal.
LEN	Maximum number of bytes to read from terminal (2-byte integer).
ACTLEN	Actual number of bytes read from terminal (2-byte integer).
TMODE	Terminal setting at the time of read (2-byte integer).

1	assume terminal is in format mode.
2	assume terminal is in unformatted mode.

LOC	Start position of write (array of two 2-byte integers). Absolute cursor addressing is not allowed in format mode. An error will be returned.

[0] [0]	home cursor before read
[-1] [0]	start from current cursor position

BC	Buffer control (2-byte integer) -- not currently used. Must initialize to zero.
TC	Terminal control (2-byte integer)--not currently used. Must initialize to zero.

This intrinsic reads a block of data from the terminal with a number of options. There are two major differences between VREADFIELDS and this procedure. First, it provides more options for reading data from the terminal. Second, data read is returned directly to the application buffer. There is no VPLUS form associated with the read.

Like the companion intrinsic, VBLOCKWRITE, this procedure is recommended only for advanced programmers who are proficient with terminal input/output. VOPENTERM must be called before using VBLOCKREAD. The keyboard must be unlocked before calling VBLOCKREAD. (Refer to keyboard unlock options in VBLOCKWRITE.) VBLOCKREAD will lock the keyboard immediately after **Enter** or a function key is pressed to ensure data integrity.

This procedure is primarily designed for unformatted reads. For users who do not use VREADFIELDS but use VBLOCKREAD to read data in format mode, the application data interpretation algorithm should accommodate both MDT (Modified Data Tag) and non-MDT inputs. When MDT is on, unmodified blanks and data are not transmitted from the terminal. Refer to the appropriate terminal reference manuals for further explanation of the MDT feature.

The following examples illustrate a call to VBLOCKREAD using common programming languages:

COBOL:

```
CALL “VBLOCKREAD” USING COMAREA @ BUF LEN ACTLEN TMODE LOC BC TC.
```

BASIC:

```
CALL VBLOCKREAD(C(*),B1$,L1,L2,M1,L(*),U1,U2)
```

FORTRAN:

```
CALL VBLOCKREAD(COMAREA,BUF,LEN,ACTLEN,TMODE,LOC,BC,TC)
```

SPL:

```
VBLOCKREAD(COMAREA,BUF,LEN,ACTLEN,TMODE,LOC,BC,TC);
```

Pascal:

```
VBLOCKREAD(COMAREA,BUF,LEN,ACTLEN,TMODE,LOC,BC,TC);
```

VBLOCKWRITE

The VBLOCKWRITE intrinsic writes a block of characters to a terminal in block mode. The syntax and parameter descriptions for this intrinsic are provided below.

```
VBLOCKWRITE {COMAREA,BUF,LEN,TMODE,LOC,TC}
```

COMAREA	The following COMAREA fields must be set before calling VBLOCKWRITE, if not already set:	
	LANGUAGE	Set to code identifying the programming language of the calling program.
	COMAREALEN	Set to total number of 2-byte words in COMAREA.
	VBLOCKWRITE may set the following COMAREA fields:	
	CSTATUS	Set to nonzero value if call is unsuccessful.
	FILERRNUM	Set to file error code if MPE file error.
BUF	Byte array containing characters to be written to the terminal.	
LEN	Number of bytes in the BUF array (2-byte integer).	
TMODE	Terminal mode (2-byte integer).	
	0	do not change terminal mode.
	1	change to format mode.
	2	change to unformatted mode.
LOC	Start position of write (array of two 2-byte integers). Absolute cursor addressing is not allowed in format mode. An error will be returned.	
	[0] [0]	home cursor before WRITE
	[x] [y]	start from absolute row x, column y. (Not allowed in format mode.)
	[-1] [0]	start from current position
	[-2] [0]	start from first available line of display memory, for example, the first available line after the end of a previous form. (Not allowed in format mode.)
TC	Terminal control (2-byte integer)	
	0	do not lock keyboard at the beginning of write; unlock at the end of write.
	1	lock keyboard at the beginning of write; unlock at the end of write.

This procedure writes the content of a user buffer to a terminal. TMODE options can be used to change the terminal to format or unformatted mode before the write. LOC options allow the programmer to specify the position of the screen where the write is to begin. Terminal control (TC) options can be used to control keyboard locking for the protection of data as it is being written to the terminal. TC = 1 is recommended for applications which do multiple writes to the terminal with no intervening reads. Procedures, such as VBLOCKREAD or

VREADFIELDS, lock the keyboard as soon as the terminal begins transmitting data when triggered by the **Enter** key or a function key.

VOPENTERM must be called before using this procedure. This procedure is intended only for advanced programmers who are proficient with terminal control operations and VPLUS terminal settings. Terminal keyboard operations, such as PREV PAGE and NEXT PAGE, can be performed programmatically by sending the appropriate escape sequences to the terminal via VBLOCKWRITE. VBLOCKWRITE can also be used to write large blocks of unformatted text or multiple report lines in between uses of predefined VPLUS forms. To ensure portability of the application from one driver to another, alteration of terminal straps using VBLOCKWRITE is not recommended. See VTURNON and VTURNOFF for information on how to switch between character mode and block mode without disturbing the screen.

The following examples illustrate a call to VBLOCKWRITE using common programming languages:

COBOL:

```
CALL “VBLOCKWRITE” USING COMAREA @BUF LEN TMODE LOC TC.
```

BASIC:

```
CALL VBLOCKWRITE(C(*),B1$,L1,M1,L(*),T1)
```

FORTRAN:

```
CALL VBLOCKWRITE(COMAREA,BUF,LEN,TMODE,LOC,TC)
```

SPL:

```
VBLOCKWRITE(COMAREA,BUF,LEN,TMODE,LOC,TC);
```

Pascal:

```
VBLOCKWRITE(COMAREA,BUF,LEN,TMODE,LOC,TC);
```

VCHANGEFIELD

Allows dynamic field attribute definition.

Syntax

```
VCHANGEFIELD {comarea,specbuffer,numentries}
```

Parameters

comarea	Must be *comarea* specified when forms file was opened with VOPENFORMF. If not already set, the following *comarea* items must be set before calling VCHANGEFIELD:

cstatus	Set to zero.
language	Set to code identifying the programming language of the calling program.
comarealen	Set to total number of two-byte words in *comarea*.

VCHANGEFIELD may set the following *comarea* items:

cstatus	Set to nonzero value if call unsuccessful.

specbuffer	A record; the specifications buffer is an array that provides VCHANGEFIELD with the information regarding the fields which are to be altered. The format of the array is as shown in Table 6-8.
numentries	A two-byte integer indicating the number of entries in the specifications buffer. The specifications buffer can contain up to three types of changes for each field in the form. That is, one call to VCHANGEFIELD can change the field type, data type and enhancement of every field in a form. For example, if a form contains four fields and an application is designed to make three changes to each field, then *numentries* should have a value of 12.

Discussion

VCHANGEFIELD alters the run-time copy of the current form. It does not modify the contents of the forms file. The next call to VGETNEXTFORM which actually retrieves a copy of the form from the forms file will reset the field specifications. The only exception to this is when two calls to VGETNEXTFORM are performed without resetting *nfname*; the second call will *not* retrieve the form definition from the forms file.

Please note that it is the responsibility of the programmer to ensure that a new data type is compatible with any initial values or processing specifications which may have been defined for the field.

If the change type selected is a toggle type, (change type of 1, 2 or 3), the change specification buffer is returned with the "old" characteristics. Hence, the next VCHANGEFIELD call with the same buffer will toggle between the characteristics specified in the first VCHANGEFIELD call and the original characteristics. To toggle between two specific characteristics, VCHANGEFIELD can be used to set a field (change type of 4, 5, or 6) to certain starting characteristics without the toggle option at first.

Table 6-8. Specifications Buffer

Data Type	Position	Description
Integer (Two-byte)	1	positive field number or negative screen order number
Integer (Two-byte)	2	Change type 1 = toggle enhancement 2 = toggle field type 3 = toggle data type 4 = change enhancement 5 = change field type 6 = change data type
Character array (4-byte)	3-4	Change Specifications Type 1, 4 values = `H, I, B, U, NONE` Type 2, 5 values = `0, D, P, R` Type 3, 6 values = `CHAR, DIG, IMP`*n*, `NUM` [*n*], `DMY, MDY, YMD`

`VCHANGEFIELD` has no effect on the security or color enhancements; an error is returned if the codes for these enhancements (`1-8` or `S`) are specified. `VCHANGEFIELD` does not alter the field error enhancement nor is this enhancement code returned when the enhancement attribute is toggled (rather the current "normal" enhancement is altered and returned if toggling).

Example

COBOL

```
CALL "VCHANGEFIELD" USING COMAREA,SPECBUF,ENTRIES.
```

BASIC

```
CALL VCHANGEFIELD(C(*),Bl,EN)
```

FORTRAN

```
CALL VCHANGEFIELD(COMAREA,SPECBUF,ENTRIES);
```

SPL/PASCAL

```
 VCHANGEFIELD(COMAREA,SPECBUF,ENTRIES);
```

VCLOSEBATCH

Closes an open batch file.

Syntax

```
VCLOSEBATCH" {comarea}
```

Parameters

comarea	Must be name of *comarea* specified in VOPENBATCH call that opened this batch file. If not already set, the following *comarea* items must be set before calling VCLOSEBATCH:

cstatus	Set to zero.
comarealen	Set to total number of two -byte words in *comarea*.

VCLOSEBATCH may set the following *comarea* items:

cstatus	Set to nonzero value if call unsuccessful.
filerrnum	Set to file error code if MPE file error.

Discussion

The open batch file identified by *comarea* is closed when this intrinsic is called. The batch file must be closed before VCLOSEFORMF is called.

Example

COBOL

```
CALL "VCLOSEBATCH" USING COMAREA.
```

BASIC

```
200 CALL VCLOSEBATCH(C(*))
```

FORTRAN

```
CALL VCLOSEBATCH(COMAREA)
```

SPL/PASCAL

```
VCLOSEBATCH(COMAREA);
```

VCLOSEFORMF

Closes an open forms file.

Syntax

`VCLOSEFORMF` *{comarea}*

Parameters

comarea	Must be *comarea* specified when forms file was opened with VOPENFORMF. If not already set, the following *comarea* items must be set before calling VCLOSEFORMF:
	cstatus — Set to zero.
	comarealen — Set to total number of two-byte words in comarea.
	VCLOSEFORMF may set the following *comarea* items:
	cstatus — Set to nonzero value if call unsuccessful.
	filerrnum — Set to file error code if MPE file error.

Discussion

The open forms file is closed when this intrinsic is executed. Once closed, the forms file is not available for further processing.

Example

COBOL

```
CALL "VCLOSEFORMF" USING COMAREA.
```

BASIC

```
200 CALL VCLOSEFORMF(C(*))
```

FORTRAN

```
CALL VCLOSEFORMF(COMAREA)
```

SPL/PASCAL

```
VCLOSEFORMF(COMAREA);
```

VCLOSETERM

Closes an open terminal file.

Syntax

`VCLOSETERM {`*`comarea`*`}`

Parameters

comarea	Must be *comarea* named when file was opened by VOPENTERM. If not already set, the following *comarea* items must be set before calling VCLOSETERM:

cstatus	Set to zero.
comareaten	Set to total number of two-byte words in *comarea*.

VCLOSETERM may set the following *comarea* items:

ctstatus	Set to nonzero value if call unsuccessful.
filerrnum	Set to file error code if MPE file error.

Discussion

The terminal file opened with the specified *comarea* is closed when this intrinsic is executed. For terminals with this feature, the local forms storage of the terminal is cleared. For additional information about about using the pseudo intrinsic .LOC to put and address into a word of the COMAREA, refer to the *COBOL II/XL Reference Manual*.

Example

COBOL

```
CALL "VCLOSETERM" USING COMAREA.
```

BASIC

```
200 CALL VCLOSETERM(C(*))
```

FORTRAN

```
CALL VCLOSETERM(COMAREA)
```

SPL/PASCAL

```
VCLOSETERM(COMAREA);
```

VERRMSG

Returns a message corresponding to the error number of an edit error or an intrinsic call error.

Syntax

VERRMSG {*comarea,buffer,buflen,actualen*}

Parameters

comarea	Must be *comarea* named when forms file was opened with VOPENFORMF. If not already set, the following *comarea* items must be set before calling VERRMSG:

language	Set to code identifying the programming language of the calling program.
comarealen	Set to total number of two-byte words in *comarea*.
errfilenum	Contains MPE file number of VPLUS error message file; should be initialized to zero so that VERRMSG can open the error message file.

(Note that *cstatus* must not be cleared before calling this intrinsic.)

VERRMSG may set the following *comarea* items:

errfilenum	If initialized to zero, VERRMSG opens VPLUS error message file and sets *errfilenum* to MPE file number of the opened file.

buffer	Character string in an application to which message is returned by VERRMSG. In order to contain the message, this buffer must be defined as at least 72 bytes long.
buflen	Two-byte integer variable set by an application to the length of the buffer.
actualen	Two-byte integer to which VERRMSG returns the number of bytes in the message sent to buffer.

Discussion

If an error occurs in an intrinsic call or is detected by a VPLUS edit, a call to VERRMSG returns the message associated with the error. For an intrinsic call error, *cstatus* is set to a nonzero value. If *cstatus* indicates an error, VERRMSG returns the text explaining the type and cause of the error. If VINITFORM, VFIELDEDITS, or VFINISHFORM detects a data error, *cstatus* is set to zero and *numerrs* is set to a nonzero value. If *numerrs* is set, VERRMSG returns a custom error message if there is one; otherwise, it returns the VPLUS error message associated with the edit error number in the VPLUS error message file. The error message, custom or VPLUS, is returned for the first field flagged in error.

The message describing the error is returned to an application in the area defined by the *buffer* parameter. The message length can be no longer than *buflen*; the actual length of the message is returned in *actualen*. You can then call VPUTWINDOW to move this message to the window area of memory for later display at the terminal.

The *errfilenum* contains the MPE file number of the VPLUS error message file. This file contains the error numbers and their associated messages for both intrinsic call and edit errors. It does not contain custom error messages, which are retrieved by VERRMSG through the form definition. VERRMSG opens this file if *errfilenum* equals zero when VERRMSG is called; it then sets *errfilenum* to the MPE file number of the file. The file is closed automatically when an application terminates.

If the VPLUS error message file cannot be opened and read by this intrinsic, default messages are generated. If *cstatus* is not equal to zero, the message is:

```
"VPLUS Error, COM'STATUS is nnn".
```

If *cstatus* is zero and *numerrs* is greater than zero, the message is:

```
"VPLUS Edit Error nnn".
```

Example

COBOL

```
CALL "VERRMSG" USING COMAREA, BUFFER, BUFLEN, ACTUALEN.
```

BASIC

```
225 CALL VERRMSG(C(*),M$,L,M)
```

FORTRAN

```
CALL VERRMSG(COMAREA,BUFFER,BUFLEN,ACTUALEN)
```

SPL/PASCAL

```
VERRMSG(COMAREA,BUFFER,BUFLEN,ACTUALEN);
```

Assume that *filerrnum* contains an MPE file error code. The calls to VERRMSG shown above return to your application a message describing the error and the length of this message in bytes.

VFIELDEDITS

Edits data entered in each field of form and, if indicated, modifies data in the data buffer. If necessary, sets error flags.

Syntax

```
VFIELDEDITS {comarea}
```

Parameters

comarea Must be *comarea* name specified when forms file was opened with VOPENFORMF. If not set already, the following *comarea* items must be set before calling VFIELDEDITS:

cstatus	Set to zero.
comarealen	Set to total number of two-byte words in *comarea*.

VFIELDEDITS may set the following *comarea* items:

numerrs	Set to total number of fields in which errors were detected.
cstatus	Set to nonzero value if call unsuccessful.
nfname	Set to new next form name if name changed by processing specifications.
repeatapp	Set to new current form code if code changed by processing specifications.
freezapp	Set to new next form code if code changed by processing specifications.

Discussion

This intrinsic checks the data content of each field. It checks that the data type and field type are correct, and if any special processing specifications were defined for this field in the Field Edit phase, it checks that the data conforms to these specifications. If any data formatting or data movement was specified for a field, VFIELDEDITS performs these functions on the data in the data buffer.

For each field that does not pass the edit checks, VFIELDEDITS sets an error flag. These error flags are used by the VPLUS intrinsics, as shown in Table 6-9. In addition, these error flags can be accessed by an application with a call to VGETFORMINFO, which retrieves a table corresponding to the current error flag settings.

Table 6-9. Actions Used by Intrinsics

Intrinsic	Action
VGETNEXTFORM	Initializes all the error flags to zero.
VFIELDEDITS VFINISHFORM VINITFORM VSETERROR	Each may set field error flags on, increment *numerrs*, and set an internal error number used by VERRMSG.
VSHOWFORM	Enhances the fields whose error flags are set.
VREADFIELDS	Resets the flags to zero after the form is displayed.
VPUTBUFFER VPUTFIELD VPUTtype	Each may clear field error flags and decrement *numerrs* if new data is entered into a field with an error.

After setting error flags for all fields with errors, VFIELDEDITS saves the error number of the first field with an error. (Fields are counted in screen order, starting at the top left and moving left to right, then top to bottom.) VFIELDEDITS sets *numerrs* to the total number of fields in which errors were found.

If requested by a call to VERRMSG, the text associated with the error number in the VPLUS error message file, or the associated custom error message, is returned to an application. If requested by a call to VPUTWINDOW, the message is copied to the window area of memory. Then, a call to VSHOWFORM can be used to display this message on the terminal screen and enhance the fields with errors.

Example

COBOL

```
CALL "VFIELDEDITS" USING COMAREA.
```

BASIC

```
150 CALL VFIELDEDITS(C(*))
```

FORTRAN

```
CALL VFIELDEDITS(COMAREA)
```

SPL/PASCAL

```
VFIELDEDITS(COMAREA);
```

VFINISHFORM

Performs any processing specifications defined for the final phase of fields editing.

Syntax

`VFINISHFORM {`*`comarea`*`}`

Parameters

comarea Must be *comarea* name specified when the forms file was opened with VOPENFORMF. If not already set, the following *comarea* items must be set before calling VFINISHFORM:

cstatus	Set to zero.
comarealen	Set to total number of two-byte words in *comarea*.

VFINISHFORM may set the following *comarea* items:

cstatus	Set to nonzero value if call unsuccessful.
numerrs	Set to total number of fields in the form in which errors were detected.
nfname	Set to name of next form if processing specifications altered form name.
repeatapp	Set to new repeat code if processing specifications altered code.
freezapp	Set to new next form code if processing specifications altered code.

Discussion

All special processing defined as part of the "finish" phase of field editing is performed by this intrinsic. Altering the next form to be displayed is a typical finish operation, and updating a save field is another. (Refer to the discussion of phases in Section 4.) Like VFIELDEDITS, VFINISHFORM sets an error flag for each field that has an error as a result of processing the form. (Refer to the VFIELDEDITS discussion.)

Example

COBOL

```
CALL "VFINISHFORM" USING COMAREA.
```

BASIC

```
160 CALL VFINISHFORM(C(*))
```

FORTRAN

```
CALL VFINISHFORM(COMAREA)
```

SPL/PASCAL

```
VFINISHFORM(COMAREA);
```

The examples above perform all finish operations on a form.

VGETARBINFO

A new VPLUS intrinsic has been added to retrieve the VPLUS ARB field mapping information. The intrinsic is named VGETARBINFO and is callable as follows:

```
      CALL “VGETARBINFO” USING VPLUS-COMAREA,
                               ARB-INFO-BUF,
                               INFO-BUF-LEN.

     Record VPPLUS-COMAREA << use existing definitions >>

      Record ARB-INFO-BUF:

      Record ARB-INFO-HEADER:

        2 byte integer NUM-OF-ENTRIES;     << input only >>
        2 byte integer ENTRY-LEN;          << input only >>
        6 byte array FORM-NAME;            << input only >>
        16 byte array FILLER;              << reserved for future use >>
        2 byte integer NUM-ARB-FLDS.       << output only >>

      Table of record ARB-INFO-DETAIL:

        2 byte integer FIELD-NUM;          << input only >>
        10 byte array ARB-DATA-TYPE;       << output only >>
        2 byte integer ARB-DATA-LEN;       << output only >>
        2 byte integer ARB-BUF-OFFSET.     << output only >>

    2 byte integer INFO-BUF-LEN.      << total 2 byte word length of >>
                                      << ARB-INFO-BUF, minimum valid >>
                                      << length is 27. >>;
```

NUM-OF-ENTRIES	indicates number of ARB-INFO-DETAIL table entries and must be zero or greater; if number of entries is zero then the intrinsic returns immediately to the application (in other words, does a no-op).
ENTRY-LEN	indicates number of ARB-INFO-DETAIL table entries and must be zero or greater; if number of entries is zero then the intrinsic returns immediately to the application (in other words, does a no-op).
FORM-NAME	contains justified, upshifted form name of 1 to 15 characters, blank terminated.
FILLER	reserved for future use; should be initialized to 16 blanks.
NUM-ARB-FLDS	at output indicates the number of ARB fields defined for the specified form. Values for NUM-ARB-FLDS range from zero (no active ARB fields in specified form ARB; all are filler), to 128 (maximum number of fields in any one form).

NOTE If this intrinsic is called for a form with zero active fields in the form’s ARB, the intrinsic continues to process all ARB-INFO-DETAIL entries, returning $NOTARBFLD for each entry.)

FIELD-NUM	is ARB-INFO-DETAIL table key and contains field number

	assigned by FORMSPEC.
ARB-DATA-TYPE	at output contains field type conversion notation from FORMSPEC, for example, CHAR, DINT, SPACK2, or token $NOTARBFLD, which indicates that the requested field does not exist within the ARB.
ARB-DATA-LEN	at output contains ARB field length in bytes (not updated if field does not exist in ARB).
ARB-BUF-OFFSET	at output contains zero relative byte offset of the ARB field (not updated if field does not exist in ARB).

VGETBUFFER

Copies entire contents of data buffer from memory to an application.

Syntax

VGETBUFFER {*comarea,buffer,buflen*}

Parameters

comarea	Must be *comarea* name specified when the forms file was opened with VOPENFORMF. If not already set, the following *comarea* items must be set before calling VGETBUFFER:

cstatus	Set to zero.
language	Set to the code identifying the programming language of the calling program.
comarealen	Set to total number of two-byte words in *comarea*. Must be at least 70 words in length if the ARB feature is used.
buffercontrol	Set bit 15 to 1 to indicate that data is to be transformed according to the ARB specifications.

VGETBUFFER may set the following *comarea* items:

cstatus	Set to nonzero value if call unsuccessful.
bufferstatus	Bit 15 set to 1 if data conversion successful.

buffer	Character string in an application to which the data in the data buffer is copied. Could also be a record describing the ARB, i.e., non-CHAR data.
buflen	Two-byte integer variable that specifies the number of bytes to be transferred to the user buffer.

Discussion

This intrinsic transfers data from the data buffer in memory to the area in an application specified by the *buffer* parameter. The data includes everything in the unprotected and display-only fields on a form. Previously, data in the data buffer was stored in the order of the fields on the form, and specific fields could be moved from the buffer with VGETFIELD or VGETtype. Now, the Application-ready Buffer (ARB) allows you to specify, using the FORMSPEC ARB feature, the order in which the application should receive the fields in the buffer, and the data type conversion to be performed on each (see the discussion on creating an ARB in Section 3). You need no longer use VGETtype to convert individual fields; a call to VGETBUFFER accomplishes the task.

The number of bytes moved from the data buffer is based on the number of bytes specified in the *buflen* parameter, or the number of bytes specified by the *dbuflen* item in the *comarea* (refer to Table 6-5), whichever is less. The *dbuflen* item is set by VGETNEXTFORM when the current form is read into memory. For example, if there are 20 bytes in the data buffer (*dbuflen* is 20), and the user requests 50 bytes in the *buflen* parameter, only 20 bytes are transferred. Conversely, if the user requests 10 bytes through the *buflen* parameter, but there are 20 bytes in the data buffer (*dbuflen* is 20), only 10 bytes are transferred.

Special Considerations

Designers using the ARB feature in VGETBUFFER should be aware that damaging run-time errors could occur if the application is inadvertently run on a system that has a VPLUS version earlier than B.05.00.

To prevent this, the designer should do three things:

1. Document the product with a clear warning that VPLUS version B.05.00 or later MUST be used.
2. Use the VPLUS intrinsic HP32209 in the code. This intrinsic checks to make sure that you are using the proper VPLUS version. If not, the application should terminate with an appropriate message.
3. Check offset 65 *(bufferstatus)* in the *comarea* on return from VGETBUFFER. Bit 15 will be set to 1 if VGETBUFFER performed the conversion successfully. In other words, the application must check both *status* and *bufferstatus* to be sure that the data was correctly converted.

Example

COBOL

```
01 ORDER-ENTRY.
    03 PART-NO PIC X(7).
    03  DESCR     PIC X(12).
    03  QTY       PIC S9(4).
    03  UNIT-PR   PIC S9(4)V9(2).
    03  TOTL-PR   PIC S9(6)V9(2).
         :
         :
CALL "VGETBUFFER" USING COMAREA, ORDER-ENTRY, DBUFLEN.
```

BASIC

```
340 L1=D1           <D1 is the dbuflen word in C
350 CALL VGETBUFFER(C(*),D$,L1)
```

FORTRAN

```
CALL VGETBUFFER(COMAREA,D1,DBFLEN)
```

SPL/PASCAL

```
BYTE ARRAY D1 (0:36) ;
   :
   :
VGETBUFFER(COMAREA,D1,DBUFLEN);
```

The examples above transfer the contents of the data buffer in memory to an application. The current value of *dbuflen* is specified as the user buffer length.

VGETFIELD

Copies contents of specified field from data buffer in memory to an application.

Syntax

```
        VGETFIELD {comarea,fieldnum,fieldbuf,buflen,actualen,nextfldnum}
```

Parameters

comarea Must be *comarea* name specified when the forms file was opened with VOPENFORMF. If not already set, the following *comarea* items must be set before calling VGETFIELD:

cstatus	Set to zero.
language	Set to the code identifying the programming language of the calling program.
comarealen	Set to total number of two-byte words in *comarea*.

VGETFIELD may set the following *comarea* items:

cstatus	Set to nonzero if call unsuccessful, or if requested field has an error, or if *fieldnum* is unacceptable.

fieldnum Two-byte integer variable containing the number assigned to the field by FORMSPEC.

fieldbuf Character string in an application to which data entered in specified field is copied.

buflen Two-byte integer variable that specifies the number of bytes in *fieldbuf*.

actualen Two-byte integer to which VGETFIELD returns the number of bytes actually moved to *fieldbuf*.

nextfldnum Two-byte integer to which VGETFIELD returns the number of the next field in screen order. If there are no more fields, zero is returned. If *fieldnum* was set to zero or a negative number by an application, this is an error. In this case, VGETFIELD returns the number of the first field in screen order in *nextfieldnum*.

Discussion

VGETFIELD transfers the contents of the field specified by *fieldnum* from the data buffer to a variable in an application. This is in contrast to VGETBUFFER which retrieves data according to the current field layout.

When considering what is transferred by VGETFIELD, keep in mind that all the fields defined for a particular form in FORMSPEC are assigned numbers. The number assigned to a field by FORMSPEC does not change regardless of any changes to the field's position in the form or to its length and does not necessarily correspond to the screen order. The field numbers on a form can only be changed with the batch command, RENUMBER, as described in Section 7. The field number must not be confused with the field's position in the data buffer, which corresponds to its position in the form according to screen order, not assigned field number.

If the number of bytes specified by *buflen* is less than the field size, the rightmost bytes are truncated. If the requested field has an error, its value is returned, but *cstatus* is set to an error number indicating the field error flag is set.

Following a successful transfer, *actualen* contains the exact number of bytes transferred to *fieldbuf*, the user buffer; *nextfldnum* is set to the number of the next field in screen order, or to zero after the last field is processed.

Note that VGETFIELD does not convert the data it moves. If you want to convert the field, you must use VGETtype, where type specifies the data type to which the field is converted.

Example

COBOL

```
DATA DIVISION.
77    ORD-LEN              PIC S9(4)COMP.
77    ITEM-LEN             PIC S9(4)COMP.
77    FIELD-NUM            PIC S9(4)COMP.
77    ACTUAL-LENGTH        PIC S9(4)COMP.
77    NEXT-FIELD           PIC S9(4)COMP.
01    ORDER-ENTRY.
      03    PART-NO        PIC X(8).
      03    UNIT-PR        PI C 9 (4) V9 (2).
      03    QUANTITY       PIC S9(4)COMP.
      03    TOTAL-PR       PIC 9(5)V9(2).
      03    PART-DESCR PIC X(12).       <- field number "2"
            :
            :
PROCEDURE DIVISION.
            :
            :
        MOVE 12 TO ITEM-LEN.
        MOVE 2 TO FIELD-NUM.
        CALL "VGETFIELD" USING COMAREA, FIELD-NUM, PART-DESCR OF ORDER-ENTR,
                                         ITEM-LEN, ACTUAL-LENGTH, NEXT-FIEL.
```

BASIC

```
350 F1=2
355 L1=12
360 CALL VGETFIELD(C(*),F1,P$,L1,A1,N1)
```

FORTRAN

```
FIELD=2
LEN=12
CALL VGETFIELD(COMAREA,FIELD,PARTDES,LEN,LENFLD,NXTFLD)
```

SPL/PASCAL

```
INTEGER
FIELD,
LEN;
BYTE ARRAY PARTDES(0:11);
  :
  :
FIELD:=2;
LEN:=12;
```

```
VGETFIELD(COMAREA,FIELD,PARTDES,LEN,ACTUAL'LEN,NEXT'FLD);
```

Assume that the contents of field number "2" is to be copied, and that the length of this field is 12 bytes. The calls shown above copy the contents of this field into a variable in an application.

VGETFIELDINFO

Returns information about specified fields to an application.

Syntax

VGETFIELDINFO {*comarea,infobuf,infobuflen*}

Parameters

comarea Must be *comarea* name specified when the forms file was opened with VOPENFORMF. If not already set, the following *comarea* items must be set before calling VGETFIELDINFO:

cstatus	Set to zero.
comarealen	Set to total number of two-byte words in *comarea*.

VGETFIELDINFO may set the following *comarea* item:

cstatus	Set to nonzero value if call unsuccessful.

infobuf A record through which you pass the request for field information and to which the intrinsic returns the specified information. This parameter must be initialized to spaces before filling in your request and calling VGETFIELDINFO. The layout of *infobuf* is shown in Table 6-10.. The three required fields of *infobuf* pass user-supplied parameters to VGETFIELDINFO as follows:

numofentries	Specifies how many fields you want information about.
entrylength	Indicates how many two-byte words of information (maximum 17 as shown in Table 6-10.) you want about each field.
formname	The name of the form that contains the fields you are inquiring about.

The rest of *infobuf* (beginning with position 11) must consist of *entrylength* number of words for each field you want information about. Thus, the total length of *infobuf*in words must be as follows:

10 + (*entrylength* * *numofentries*)

You may pass one or more of the three permissible keys, which must be passed in the position indicated in the table. Remember that *infobuf* must be initialized to spaces before filling in the parameters and any key.

infobuflen Two-byte integer variable set to the number of two-byte words in *infobuf*.

Discussion

This intrinsic accesses an internal table and places information about one or more fields into *infobuf*. You tell VGETFIELDINFO how many fields, how much information about each field, and the name of the form containing the fields.

Table 6-10. Field Information Buffer

Data Type	Position	Contents	Comments
Integer (Two-byte)	1	*numofentries*	Required
	2	*entrylength*	Required
Character Array (16-byte)	3-10	*formname*	Required; second byte of position 10 is unused
	11-18	Field name	Permissible key; last byte of position 18 is unused
Integer (Two-byte)	19	Field number according to screen order	Permissible key
	20	Field number	Permissible key; in order of creation
	21	Field length	In bytes
	22	Position of field in data buffer	In bytes, offset from zero
Character Array (4-byte)	23-24	Field enhancement	Combination of I, H, U, B, 1-8, or NONE.
	25-26	Data type of field	May be CHAR, DIG, IMP*n*, date (MDY, DMY, YMD), or NUM[n].
Character (Two-byte)	27	Field type	First byte of position 27; may be O, R, P or D.

Passing INFOBUF Without Entering Keys

Keys are optional. If you do not supply a key to indicate the first field you want information about, VGETFIELDINFO starts with the first field not already reported on by the current call.

As an example, suppose you want information about 10 fields, and you enter screen order numbers 3, 5, and 6 as keys. VGETFIELDINFO returns information about fields 3, 5, 6, and 7 through 13. If you enter no keys, VGETFIELDINFO returns information about fields 1 through 10.

Table Wraparound

VGETFIELDINFO starts over with the first field in the form if you request information about more fields than are in the form or, if when you pass a key, your request goes beyond the end of the form.

Suppose a form has 12 fields. If you request information about 10 of the fields and enter screen order number 8 as a key, VGETFIELDINFO returns information about fields 8 through 12, and then goes to the beginning of the form and returns information about fields 1 through 5.

Example

COBOL

```
DATA DIVISION.
       :
       :
WORKING-STORAGE SECTION.
01 INFOBUF.
    05    NUMBER-OF-ENTRIES                          PIC S9(4) COMP.
    05    ENTRY-LENGTH                               PIC S9(4) COMP.
    05    FORM-NAME                                  PIC X(15).
    05    FILLER                                     PIC X.
    05    ENTRY-TABLE OCCURS 10 TIMES.
          10  FIELD-NAME                             PIC X(15).
          10 FILLER                                  PIC X.
          10  SCREEN-ORD-NUM                         PIC S9(4) COMP.
          10  FIELD-NUM                              PIC S9(4) COMP.
          10  FIELD-LENGTH                           PIC S9(4) COMP.
          10  FIELD-POSITION                         PIC S9(4) COMP.
          10 FIELD-ENHANCE                           PIC X(4).
01 INFOBUFLEN                                        PIC S9(4) COMP VALUE 80.
       :
       :
PROCEDURE DIVISION.
       :
       :
    MOVE SPACES TO INFOBUF.
       :
    MOVE 5 TO NUMBER-OF-ENTRIES.
    MOVE 14 TO ENTRY-LENGTH.
    MOVE "FORM1                " TO FORM-NAME.
    MOVE 2 TO SCREEN-ORD-NUM.
    CALL "VGETFIELDINFO" USING COMAREA, INFOBUF, INFOBUFLEN.
```

The example shown above illustrates the data declaration of *infobuf* and *infobuflen* and the passing of parameters to VGETFIELDINFO. Note that before the intrinsic is called, *infobuf* is initialized to spaces (*not* zeros). The intrinsic copies 14 two-byte words of information about fields 2 through 6 of form FORM 1 into *infobuf*.

VGETFILEINFO

Returns information about forms file to an application.

Syntax

`VGETFILEINFO {`*`comarea,infobuf,infobuflen`*`}`

Parameters

comarea	Must be *comarea* name specified when the forms file was opened with VOPENFORMF. If not already set, the following *comarea* items must be set before calling VGETFILEINFO:
	cstatus Set to zero.
	comarealen Set to total number of two-byte words in *comarea*.
	VGETFILEINFO may set the following *comarea* item:
	cstatus Set to nonzero value if call unsuccessful.
infobuf	A record through which you pass the request for forms file information and to which the intrinsic returns the specified information. This parameter must be initialized to spaces before filling in your request and calling VGETFILEINFO. The layout of *infobuf* is shown in Table 6-11. The two required fields of *infobuf* pass user-supplied parameters to VGETFILEINFO as follows:
	numofentries Specifies how many times you want the information repeated (information is returned only about the forms file associated with *comarea*; if you specify a number greater than one, you get duplicate information).
	entrylength Indicates how many two-byte words of information (maximum 19 as shown in Table 6-11.) you want about the file.
	Remember that *infobuf* must be initialized to spaces before filling in the parameters.
infobuflen	Two-byte integer variable set to number of two-byte words in *infobuf*.

Discussion

This intrinsic accesses an internal table and places information about the open forms file into *infobuf.* You tell VGETFILEINFO how much information you want about the file.

Table 6-11. File Information Buffer

Data Type	Position	Contents	Comments
Integer (Two-byte)	1	*numofentries*	Required
	2	*entrylength*	Required
Integer Array (Four-byte)	3-4	File version number	A date/time stamp, recorded at last forms file compile.
Integer (Two-byte)	5	Number of forms in file	
	6	Maximum number of fields	In any one form
	7	Maximum data buffer size in two-byte words	Of any one form, plus up to one additional two-byte word.
	8	Number of save fields in file	
Character Array	9-16	Name of head form in file	Last byte of position 16 is unused
	17-18	Global error enhancement	Combination of `I, H, U, B, 1-8,` or NONE.
	19-20	Global window enhancement	Combination of `I, H, U, B, 1-8,` or NONE.
Integer (Two-byte)	21	Position of window	Line number, `0-23`. The line closest to the top of the form is 0; 255 means no window.

Example

COBOL

```
DATA DIVISION.
              :
              :
WORKING-STORAGE SECTION.
01 INFOBUF.
    05   NUMBER-OF-ENTRIES                         PIC S9(4) COMP.
    05   ENTRY-LENGTH                              PIC S9(4) COMP.
    05   ENTRY-TABLE OCCURS 1 TIMES.
         10 FILE-VERSION                           PIC S9(8) COMP.
         10 NUMBER-OF-FORMS                        PIC S9(4) COMP.
         10 MAX-FIELDS                             PIC S9(4) COMP.
         10 MAX-BUFSIZE                            PIC S9(4) COMP.
         10 SAVE-FIELDS                            PIC S9(4) COMP.
         10 HEAD-FORM                              PIC X(15).
         10 FILLER                                 PIC X.
         10 ERROR-ENHANCE                          PIC X(4).
         10 WINDOW-ENHANCE                         PIC X(4).
         10 WINDOW-POSITION                        PIC S9(4) COMP.
01 INFOBUFLEN                                      PIC S9(4) COMP VALUE 21.
      :
      :
PROCEDURE DIVISION.
      :
      :
    MOVE SPACES TO INFOBUF.
      :
      :
    MOVE 1 TO NUMBER-OF-ENTRIES.
    MOVE 19 TO ENTRY-LENGTH.
    CALL "VGETFILEINFO" USING COMAREA, INFOBUF, INFOBUFLEN.
```

The example shown above illustrates the data declaration of *infobuf* and *infobuflen* and the passing of parameters to VGETFILEINFO. Note that before the intrinsic is called, *infobuf* is initialized to spaces (*not* zeros). The intrinsic copies 19 two-byte words of information about the open forms file into *infobuf*.

VGETFORMINFO

Returns information about specified forms to an application.

Syntax

VGETFORMINFO {*comarea*,*infobuf*,*infobuflen*}

Parameters

comarea	Must be *comarea* name specified when the forms file was opened with VOPENFORMF. If not already set, the following *comarea* items must be set before calling VGETFORMINFO:
	cstatus Set to zero.
	comarealen Set to total number of two-byte words in *comarea*.
	VGETFORMINFO sets the following field:
	cstatus Set to nonzero value if call unsuccessful.
infobuf	A record through which you pass the request for form information and to which the intrinsic returns the specified information. This parameter must be initialized to spaces before filling in your request and calling VGETFORMINFO. The layout of *infobuf* is shown in Table 6-12. The two required fields of *infobuf* pass user-supplied parameters to VGETFORMINFO as follows:
	nurmofentries Specifies how many forms you want information about.
	entrylength Indicates how many two-byte words of information (maximum 36 as shown in Table 6-12.) you want about each form.
	The rest of *infobuf* (beginning with word 3) must consist of *entrylength* number of two-byte words for each form you want information about. Thus, the total length of *infobuf* in two-byte words must be as follows;

```
2 + (entrylength * numofentries)
```

	You may pass one or both of the two permissible keys, which must be passed in the position indicated in the table. Remember that *infobuf* must be initialized to spaces before filling in the parameters and any key.
infobuflen	Two-byte integer variable set to the number of two-byte words in *infobuf*.

Discussion

This intrinsic accesses an internal table and places information about forms into *infobuf*. You tell VGETFORMINFO how many forms and how much information you want about each form.

Table 6-12. Form Information Buffer

Data Type	Position	Contents	Comments
Integer (Two-byte)	1	*numofentries*	Required
	2	*entrylength*	Required
Character Array (16-byte)	3-10	Form name	Permissible key; last byte of position 10 is unused (15 bytes maximum).
Integer (Two-byte)	11	Form number	Permissible key; forms are ordered within the forms file in alphabetic order, thus this argument reflects the order of the form within the file, not a permanently assigned form numeric identifier. As a result, the number may change if the forms file is recompiled.
	12	Number of fields in form	
	13	Data buffer length in bytes	
Character Array (16-byte)	14-21	Name of next form.	Last byte of position 21 is unused.
Character (Two-byte)	22	Repeat option.	First byte of position 22; may be N, A, or R.
	22	Next form option	Last byte of position 22; may be C, A, or F.
Integer Array (32-byte)	23-38	Fields in error.	Bit map; bit is on if field is in error.

Passing INFOBUF Without Entering Keys

Keys are optional. If you do not supply a key to indicate the first form you want information about, VGETFORMINFO starts with the first form not already reported on by the current call.

As an example, suppose you want information about 3 forms, and you enter form number 6 as a key. VGETFORMINFO returns information about forms 6, 7, and 8. If you enter no keys, VGETFORMINFO returns information about forms 1 through 3.

Table Wraparound

VGETFORMINFO starts over with the first form in the file if you request information about more forms than are in the file or, if when you pass a key, your request goes beyond the end of the file.

Suppose a file has 10 forms. If you request information about 5 of the forms and enter form number 8 as a key, VGETFORMINFO returns information about forms 8 through 10, and then goes to the beginning of the file and returns information about forms 1 and 2.

Error Determination

To enable applications to determine which fields are in error for a given form at run-time, VGETFORMINFO optionally returns a 32-byte bit map containing these error flags to show which fields failed the edit checks.

Each field in a form is represented by a bit in the bit map according to its field creation number. The bit map is a 32-byte logical array to accommodate the maximum of 255 field numbers. The most significant (left most) bit of the first byte in the bit map array represents the field with the field number equal to 1 as assigned by FORMSPEC. For example, the field assigned a field number of 17 would be represented by the most significant bit of the third byte in the array. If a field is in error, the corresponding bit representing that field will be set to a one. Otherwise, the value will be zero.

The bit map is valid for the current form only; information for the current form must be requested by form name key. If the bit map is requested for any other form, it will contain all zeros. Therefore, a value of zero should not be interpreted to mean the field is not in error for the current form, (i.e., the last form retrieved by VGETNEXTFORM). Furthermore, bit positions representing nonexistent field numbers for a given form will contain a value of zero.

To retrieve the bit map, the user-supplied parameter *entrylength* in *infobuf* should contain a value of 36. This is to inform VPLUS to supply 36 two-byte words of information pertaining to the form requested. The last 16 two-byte words will contain the bit map indicating which fields are in error.

The *infobuflen* parameter normally passed to VGETFORMINFO is calculated using the *entrylength* of 36. To request the bit map for the current form, the calculation is as follows:

2 + (*entrylength* * *numofentries*)

where *entrylength* is equal to 36 and *numofentries* is equal to 1. Thus, the *infobuflen* parameter should contain a value of 38 when calling VGETFORMINFO.

For languages which provide bit manipulation capability, the bits can be shifted left or right to determine which bits are on. For other languages, a mask may be used with the bit map to determine whether a bit is on or off. An intrinsic called BITMAPCNV is available to help decode and set bits in the bit map. Refer to Appendix I for more information.

Example

COBOL

```
DATA DIVISION.
       :
WORKING-STORAGE SECTION.
01 INFOBUF.
     05   NUMBER-OF-ENTRIES                       PIC S9(4) COMP.
     05   ENTRY-LENGTH                            PIC S9 (4) COMP.
     05   ENTRY-TABLE OCCURS 9 TIMES.
          10  FORM-NAME                           PIC X15.
          10  FILLER                              PIC X.
          10  FORM-NUMBER                         PIC S9(4) COMP.
          10  NUMBER-OF-FIELDS                    PIC S9(4) COMP.
          10  BUF-LENGTH                          PIC S9(4) COMP.
          10  NEXT-FORM                                  PIC X(15).
          10  FILLER                                     PIC X.
          10  REPEAT-OPTION                              PIC X.
          10  NFORM-OPTION                               PIC X.
01 INFOBUFLEN                                            PIC S9(4) COMP VALUE 62.
       :
PROCEDURE DIVISION.
       :
     MOVE SPACES TO INFOBUF.
       :
     MOVE 3 TO NUMBER-OF-ENTRIES.
     MOVE 20 TO ENTRY-LENGTH.
     MOVE "FORM1              " TO FORM-NAME.
     CALL "VGETFORMINFO" USING COMAREA, INFOBUF, INFOBUFLEN.
```

The example shown above illustrates the data declaration of *infobuf* and *infobuflen* and the passing of parameters to VGETFORMINFO. Note that before the intrinsic is called, *infobuf* is initialized to spaces (*not* zeros). The intrinsic copies 20 two-byte words of information about 3 forms in collating sequence order, beginning with the form named FORM 1, into *infobuf*.

VGETKEYLABELS

Retrieves global or current form function key labels.

Syntax

```
        VGETKEYLABELS {comarea,formnorglob,numoflabels,labels}
```

Parameters

comarea	Must be *comarea* name specified when the forms file was opened with VOPENFORMF. If not already set, the following *comarea* items must be set before calling VGETKEYLABELS:

cstatus	Set to zero.
language	Set to the code identifying the programming language of the calling program.
comarealen	Set to total number of two-byte words in *comarea*.

VGETKEYLABELS may set the following *comarea* item:

cstatus	Set to nonzero value if call unsuccessful.

formorglob Two-byte integer specifying which type of labels are to be retrieved.

- `0` - Retrieve global labels
- `1` - Retrieve current form labels

numflabels Two-byte integer indicating how many labels are to be retrieved. This value must be from 1 to 8, inclusive.

labels A byte array in which the labels are passed back to an application. The length of the array must be at least *numflabels* * 16. (Each label is 16 bytes long.)

Discussion

This intrinsic is used to copy global or current form function key labels that have been specified with FORMSPEC, VSETKEYLABEL, or VSETKEYLABELS into an application. Labels for some or for all eight function keys can be copy. The *labeloption* must be set to one prior to VOPENFORMF.

When this intrinsic is called, any active labels are retrieved from the key label buffers in memory into an application. If there are no active labels, a buffer of spaces is returned. The labels are displayed by calling VSHOWFORM.

Example

COBOL

```
77 FORM-OR-GLOB      PIC S9(4)COMP.
77 NUM-OF-LABELS     PIC S9(4)COMP.
77 KEY-LABELS        PIC X(32).
…
MOVE 1 TO FORM-OR-GLOB.
MOVE 2 TO NUM-OF-LABELS.
```

```
CALL "VGETKEYLABELS" USING COMAREA,FORM-OR-GLOB,NUM-OF-LABELS,KEY-LABELS.

BASIC
10 INTEGER F,N
20 DIM L$[32]
30 F=1
40 N=2
50 CALL VGETKEYLABELS(C[*],F,N,L$)

FORTRAN
CHARACTER*32 LABELS
INTEGER*2 FORMORGLOB,NUMLABELS
FORMORGLOB=1
NUMLABELS=2
CALL VGETKEYLABELS(COMAREA,FORMORGLOB,NUMLABELS,LABELS)

SPL/PASCAL
INTEGER
   FORM'OR'GLOB,
   NUM'OF'LABELS;
BYTE ARRAY
   LABELS(0:31);
…
FORA'OR'GLOB: =1;
NUM'OF'LABELS:=2;
VGETKEYLABELS(COMAREA,FORM'OR'GLOB,NUM'OF'LABELS,LABELS);
```

VGETLANG

Returns the native language ID assigned to the forms file. For more information on Native Language Support, see Section 8.

Syntax

```
        VGETLANG {comrea, langnum}
```

Parameters

comarea	Must be *comarea* name specified when the forms file was opened with VOPENFORMF. If not already set, the following *comarea* items must be set before calling VGETLANG:
	cstatus Set to zero.
	comarealen Set to total number of two-byte words in *comarea*.
	VGETLANG may set the following *comarea* items:
	cstatus Set to nonzero value if call is unsuccessful
langnum	Two-byte integer variable to which VGETLANG returns the native language ID assigned to the forms file.

Discussion

The forms file must be opened before calling VGETLANG. Otherwise, *cstatus* returns a nonzero value. The native language ID returned is always the number assigned to the forms file on the Terminal/Language Menu in FORMSPEC. If the international language ID (-1) is assigned, which means that VSETLANG may be used to specify the native language ID when the application executes, the native language ID returned by VGETLANG will be the number assigned to the forms file (-1), *not* the number specified with VSETLANG.

Example

COBOL

```
CALL "VGETLANG" USING COMAREA, LANGNUM.
```

BASIC

```
120 CALL VGETLANG(C(*),L)
```

FORTRAN

```
CALL VGETLANG(COMAREA,LANGNUM)
```

SPL/PASCAL

```
VGETLANG(COMAREA,LANGNUM);
```

VGETNEXTFORM

Reads the next form from an open forms file into the form definition area of memory.

Syntax

```
        VGETNEXTFORM {comarea}
```

Parameters

comarea Must be *comarea* name specified when the forms file, from which the form is to be retrieved, was opened with VOPENFORMF. If not already set, the following *comarea* items must be set before calling VGETNEXTFORM:

cstatus	Set to zero.
comarealen	Set to total number of two-byte words in *comarea*.
cmode	Set to zero.
nfname	Set to name of next form, or $END, $HEAD, $RETURN, or $REFRESH. See "Communication Area" earlier in this section for more information. (May also be set by prior call to VGETNEXTFORM, VOPENFORMF, VREADBATCH, VINITFORM, VFIELDEDITS, or VFINISHFORM.)
repeatapp	Set to 1 if current form is to be repeated, to 2 if repeated and appended. Note that *repeatapp* must be set to zero in order to retrieve and display the next form. (Also may be set by a prior call to VGETNEXTFORM, VINITFORM, VFIELDEDITS, or VFINISHFORM.)
freezapp	Set to 1 if next form is to be appended to current form, to 2 if current form is to be frozen before next form is appended. Set to zero in order to clear the current form before displaying the next form. (Also may be set by prior call to VGETNEXTFORM, VINITFORM, VFIELDEDITS, or VFINISHFORM.)

VGETNEXTFORM sets the following items in *comarea*:

numerrs	Set to zero.
cfname	Set to name of form just read from file.
cstatus	Set to nonzero value if call unsuccessful.
filerrnum	Set to file error code if MPE file error.
multiusage	Set to 1 if current form is child or sibling of previous form, zero otherwise.

In addition, if a new form has been retrieved (*repeatapp* is zero), VGETNEXTFORM sets the following items:

cfnumlines	Set to number of lines in form just read (now the current form).
nfname	Set to name of next form to be read from file. See

	Discussion below for more information.
repeatapp	Set to value read from forms file for this (current) form.
freezapp	Set to value read from forms file for this (current) form.
dbuflen	Set to length (in bytes) of the current form just read from the forms file.

Discussion

VGETNEXTFORM checks the value of *repeatapp* passed in *comarea*. If this value indicates the current form is to be repeated, or repeated and appended to itself, it does not read the next form, nor does it update the values of *cfnumlines*, *nfname*, *repeatapp*, *freezapp*, or *dbuflen*. Note that a repeating form is repeated until repeatapp is set to zero, either by an application or, for ENTRY, when the user presses NEXT FORM to request the next form, or by the FORMSPEC processing language. If the current form is not to be repeated, VGETNEXTFORM checks *nfname* to determine which form to read from the forms file.

Example

COBOL

```
CALL "VGETNEXTFORM" USING COMAREA.
```

BASIC

```
11 CALL VGETNEXTFORM(C(*))
```

FORTRAN

```
CALL VGETNEXTFORM(COMAREA)
```

SPL/PASCAL

```
VGETNEXTFORM(COMAREA);
```

The examples above call VGETNEXTFORM to retrieve the next form from the forms file and reset the *comarea* according to the values in the next form.

VGETSAVEFIELD

Copies contents of the specified save field from save field buffer in memory to an application.

Syntax

`VGETSAVEFIELD {`*`comarea,sfname,sfbuf,buflen,actualen`*`}`

Parameters

`comarea`	must be *comarea* name specified when the forms file was opened with `VOPENFORMF`. If not already set, the following *comarea* items must be set before calling `VGETSAVEFIELD`:

`cstatus`	set to zero.
`language`	set to the code identifying the programming language of the calling program.
`comarealen`	set to total number of two-byte words in *comarea*. Must be at least 70 words in length.

`VGETSAVEFIELD` may set the following *comarea* item:

`cstatus`	set to nonzero value if call is unsuccessful.

`sfname`	is a character string in the application which contains the name of the save field.
`sfbuf`	is a character string in the application to which the contents of the save field is copied.
`buflen`	is a two-byte integer variable which specifies the number of bytes in *sfbuf*.
`actualen`	is a two-byte integer variable which specifies the number of bytes actually moved to *sfbuf*.

Discussion

`VGETSAVEFIELD` transfers the contents of the save field specified by *sfname* to a variable in an application.

If the number of bytes specified by *buflen* is less than the length of the save field, the right most bytes are truncated. Following a successful transfer, *actualen* is set to the actual number of bytes transferred to *sfbuf*.

`VGETSAVEFIELD` does not convert the data moved to the application variable.

Example

```
COBOL
CALL "VGETSAVEFIELD" USING COMAREA SFNAME SFBUF BUFLEN ACTLEN.

SPL
VGETSAVEFIELD(COMAREA,SFNAME,SFBUF,BUFLEN,ACTLEN);
```

These calls will transfer the contents of the save field specified by SFNAME to SFBUF. If the call is successful, ACTLEN will contain the exact number of bytes transferred.

VGETSCPDATA

Returns information about the cursor position by field number and row and column on a VPLUS screen.

Syntax

VGETSCPFIELD {*comarea, fieldnum, screenrow, screencol*}

Parameters

comarea	Must be *comarea* name specified when the forms file was opened with VOPENFORMF. If not already set, the following *comarea* items must be set before calling VARMSCP:
	cstatus — Set to zero.
	comarealen — Set to total number of two-byte words in *comarea*. Must be at least 70 words in length.
fieldnum	Two-byte integer variable to which VGETSCPFIELD returns the field number of the field where the cursor was last positioned when the read terminated.
screenrow	Two-byte integer variable to which VGETSCPDATA returns the physical row number on the screen where the cursor was last positioned when the read terminated.
screencol	Two-byte integer variable to which VGETSCPDATA returns the physical column number on the screen where the cursor was last positioned when the read terminated.

Discussion

When VREADFIELDS terminates, the cursor position on the screen is tracked and retrieved by VGETSCPFIELD. The information contains the field number of the field in which the cursor was positioned when the read was terminated. No cursor position information is available if a VREADFIELDS retry occurs.

VGETSCPDATA returns the physical position of the cursor on the screen by row and column number. Rows are numbered from top to bottom and columns are numbered from left to right.

NOTE The row and column information returned by VGETSCPDATA is *raw* physical information and may not directly correspond to actual elements of the user interface.

Like VGETSCPFIELD, VGETSCPDATA returns a -1 in *fieldnum* if:

- The cursor was not positioned within a field when the read terminated.
- No cursor position information is available when VGETSCPDATA is called.
- The cursor is positioned within a multi-line field **and** beyond the first line of the field.

VGETSCPDATA should be called *after* each VREADFIELDS since it first retrieves the cursor position information, then deletes the cursor position information upon procedure completion.

Example

```
COBOL
CALL “VGETSCPDATA” USING COMAREA
FIELDNUM SCREENROW SCREENCOL.

SPL
VGETSCPDATA (COMAREA,FIELDNUM,
SCREENROW,SCREENCOL);
```

These examples return the cursor position on the last read in the *fieldnum*, *screenrow and screencol* variables.

VGETSCPFIELD

Returns information about the cursor position by field number on a VPLUS screen.

Syntax

```
VGETSCPFIELD {comarea,fieldnum}
```

Parameters

comarea	Must be *comarea* name specified when the forms file was opened with VOPENFORMF. If not already set, the following *comarea* items must be set before calling VARMSCP:
	cstatus — Set to zero.
	comarealen — Set to total number of two-byte words in *comarea*. Must be at least 70 words in length.
fieldnum	Two-byte integer variable to which VGETSCPFIELD returns the field number of the field where the cursor was last positioned when the read terminated.

Discussion

When VREADFIELDS terminates, the cursor position on the screen is tracked and retrieved by VGETSCPFIELD. The information contains the field number of the field in which the cursor was positioned when the read was terminated. No cursor position information is available if a VREADFIELDS retry occurs.

VGETSCPFIELD returns a -1 in *fieldnum* if:

- The cursor was not positioned within a field when the read terminated.
- No cursor position information is available when VGETSCPFIELD is called.
- The cursor is positioned within a multi-line field **and** beyond the first line of the field.

VGETSCPFIELD should be called *after* each VREADFIELDS since it first retrieves the cursor position information, then deletes the cursor position information upon procedure completion.

Example

```
COBOL
CALL “VGETSCPFIELD” USING COMAREA FIELDNUM.

SPL
VGETSCPFIELD(COMAREA,FIELDNUM;
```

These examples return the cursor position on the last read in the *fieldnum* variable.

VGETtype

Copies character coded data contents from data buffer into an application, converting numeric value to specified type.

Syntax

```
VGETtype {comarea,fieldnum,variable} [{,numdigits,decplaces}]*
```

Parameters

type The *type* in VGETtype indicates that this intrinsic may be specified as:

VGETINT	converts value to two-byte integer
VGETDINT	converts value to four-byte integer
VGETREAL	converts value to four-byte real value
VGETLONG	converts value to eight-byte long value
*VGETPACKED	converts value to packed decimal format (COBOL *usage is comp-3* format); this intrinsic has two extra parameters, *numdigits* and *decplaces*. Both are two-byte integer variables that contain the number of digits and number of decimal places, respectively, specified by the COBOL *usage is comp-3* data declaration.
VGETZONED	converts value to zoned decimal format (COBOL) default format); has the parameters *numdigits* and *decplaces*, which are two-byte integer variables that contain the number of digits and number of decimal places, respectively, specified by the COBOL data declaration.
VGETYYMMDD	converts date to a six-byte character value

```
For example:  CALL "VGETINT" USING COMAREA,FIELDNUM,VARIABLE

CALL "VGETPACKED" USING
COMAREA,FIELDNUM,VARIABLE,NUMDIGITS,DECPLACES
```

comarea Must be *comarea* name specified when the forms file was opened with VOPENFORMF. If not already set, the following *comarea* items must be set before calling VGETtype:

cstatus	Set to zero.
comrealen	Set to total number of two-byte words in *comarea*.

VGETtype may set the following *comarea* items:

cstatus	Set to nonzero value if call unsuccessful or if requested field has an error.

fieldnum Two-byte integer variable containing the number assigned to the field by FORMSPEC.

variable Variable within application, of type specified in VGETtype, into which converted value is placed.

Discussion

If there is no ARB, all data is read from the unprotected fields on the screen as character strings and concatenated to create a data buffer. The VGETBUFFER intrinsic copies the entire buffer to an application program or VGETFIELD can be used to obtain the contents of an individual field.

NOTE If you are using VGETBUFFER in conjunction with an ARB, you do not need to use VGETtype. VGETBUFFER performs all the required conversions on the screen data in the buffer.

You can use both VGETBUFFER with an ARB and VGETtype calls in the same program: the *buffercontrol* setting in the *comarea* that controls ARB processing can be switched on or off for each form.

The VGETtype intrinsics, VGETINT, VGETDINT, VGETREAL, VGETLONG, VGETYYMMDD, VGETPACKED, and VGETZONED, have been provided to perform conversion from character coded data to the seven indicated data types. This intrinsic copies the contents of the field identified by its field number from the data buffer. (Note that this field number is a unique number assigned to each field by FORMSPEC and is totally independent of the field position in the data buffer.) The field's value must be numeric, but its data type need not be. That is, numbers in a character type field can be converted.

The numeric value, stored in the buffer in character coded form, is converted to the specified type and then copied to the variable in the application. (Refer to Table 6-13. for the format of each type.) If errors occur during conversion, *cstatus* is set to an error code. If the requested field has an error, its value is moved to the variable, but *cstatus* is set to an error code.

The default behavior for VGETPACKED and VGETZONED is to produce an unsigned result. If a signed result is desired, the negative of the number of digits in the receiving variable should be passed in the “numdigits” variable. For example, if the number of digits in the receiving variables is 6, and a signed result is desired, then -6 should be placed in “numdigits”.

Table 6-13. Numeric Type Conversions

Type	Format
INT	Two-byte word fixed-point; twos complement representation of positive and negative values; range from -32768 through +32767.
DINT	Four-byte word fixed-point format; twos complement representation of positive and negative range between approximately -2 billion and +2 billion. (Not used by BASIC programs.)
REAL	Four-byte word floating-point format with sign bit in bit 0; an exponent (biased by +256) in bits 1 through 9, and a positive fraction in the remaining 22 bits in HP 3000 format. (Not used by COBOL programs.)

Table 6-13. Numeric Type Conversions

Type	Format
LONG	Floating -point format using eight-bytes; sign bit in bit 0; an exponent (biased by +256) in bits 1 through 9, and a positive fraction in the remaining 54 bits in HP 3000 format. (Not used by COBOL programs.)
PACKED	COBOL only; *comp-3*. See *COBOL II/3000 Reference Manual*.
ZONED	COBOL only. See *COBOL II/3000 Reference Manual*.
YYMMDD	Six-byte ASCII, in format YYMMDD; for example, 870623.

The following chart will help you correlate VPLUS data types with VGETtype intrinsics and programming language data types.

Table 6-14. Correlation of VGETtype with the Data Types for each Language

Language	Intrinsic	Data Type
FORTRAN	VGETINT VGETDINT VGETREAL VGETLONG	Integer*2 Integer*4 Real Double Precision
BASIC	VGETINT VGETREAL VGETLONG	Integer Real Long
SPL	VGETINT VGETDINT VGETREAL VGETLONG	Integer Double Real Long
COBOL	VGETINT VGETDINT	S9-S9(4)COMP S9(5)-S9(9)COMP
PASCAL	VGETINT VGETDINT VGETREAL VGETLONG	Subrange -32768..32767 Integer Real Longreal

1. If errors occur during conversion, the two-byte status word in the communications area is set to an error value.
2. If the requested field has been flagged as having an error (perhaps by VFIELDEDITS or VSETERROR), the conversion is performed, but the two-byte status word is also set to an error value.
3. An attempt to convert a number larger than 32767 using VGETINT returns an error value (504) in the two-byte status word and leaves the receiving value unchanged.
4. All numeric separators are stripped before conversion is performed.
5. Fields of type CHAR may be converted as long as they contain numeric characters (including signs and separators). Otherwise, an error value is returned in the two-byte

status word.

6. VGETINT and VGETDINT only convert the integer portion of a given field. The fractional portion is truncated before conversion. Remember that in a field of type IMPn, the rightmost "n" characters are treated as a fraction.
7. Note that if VGETFIELD is used to pass a field containing a decimal point to a COBOL program, the decimal point is also passed and no arithmetic may be performed on the field.
8. Negative numbers can be zoned correctly for COBOL only by using the VGETINT and VGETDINT intrinsics. VGETBUFFER and VGETFIELD transfer the negative sign, but COBOL treats the value as positive, ignoring the sign character. An EXAMINE statement using TALLY can determine that the negative sign is present and then the program can treat the value accordingly.
9. Normal rules of truncation in COBOL are followed. For example, conversion of 12345 using VGETINT with a receiving field of S9(4) truncates to the value of 2345.
10. VGETINT may be used to convert positive integers to type LOGICAL in SPL.

Example

COBOL

```
77 FIELD-NUM            PIC S9(4)COMP.
77 COUNT                PIC S9(4)COMP.
    :
MOVE 5 TO FIELD-NUM
CALL "VGETINT" USING COMAREA, FIELD-NUM, COUNT.
```

BASIC

```
210 F1=5
220 CALL VGETINT(Cl(*),F1,C)
```

FORTRAN

```
FIELD=5
CALL VGETINT(COMAREA,FIELD,K1)
```

SPL/PASCAL

```
INTEGER
     FIELD,
     COUNT;
     :
FIELD:=5;
VGETINT(COMAREA,FIELD,COUNT);
```

The calls in this example convert a value contained in field 5 in the data buffer to two-byte integer representation and store it in an application.

VGETYYYYMMDD

Converts data in the data buffer to a eight-byte character value and copies it to an application.

Syntax

`VGETYYYYMMDD {comarea,fieldnum,variable}`

Parameters

comarea	must be *comarea* name specified when the forms file was opened with VOPENFORMF. If not already set, the following *comarea* items must be set before calling VGETYYYYMMDD:
	`cstatus` — set to zero.
	`comarealen` — set to total number of two-byte words in *comarea*. Must be at least 70 words in length.
	VGETYYYYMMDD may set the following *comarea* item:
	`cstatus` — set to nonzero value if call is unsuccessful or requested field has an error.
fieldnum	is a two-byte integer variable which specifies the number assigned to the field by FORMSPEC
variable	is a character string in the application into which the converted value is placed

Discussion

VGETYYYYMMDD transfers the contents of the field specified by *fieldnum* to a variable in an application. The contents of the field are stored in a data buffer from which the value is taken. This value is converted to YYYYMMDD format and the converted value is placed in the application variable. The YYYYMMDD format is an 8-byte ASCII value, for example, 19961225.

If errors occur during conversion, *cstatus* is set to an error code. If the requested field has an error, its value is moved to the variable but *cstatus* is set to an error code.

Refer to the VGETtype intrinsic description in the *Data Entry and Forms Management System VPLUS Reference Manual* for related information.

Example

```
COBOL
CALL "VGETYYYYMMDD" USING COMAREA FIELDNUM VARIABLE.

SPL
VGETYYYYMMDD(COMAREA,FIELDNUM,VARIABLE);
```

These calls will convert the contents of the data buffer corresponding to the field specified by FIELDNUM and place the converted value into VARIABLE.

VINITFORM

Initializes fields in data buffer according to specifications defined in the initialize phase of field definition. Both VOPENFORMF and VGETNEXTFORM must have been executed successfully prior to calling VINITFORM.

Syntax

```
          VINITFORM {comarea}
```

Parameters

comarea	Must be *comarea* name specified when the forms file was opened with VOPENFORMF. If not already set, the following *comarea* items must be set before calling VINITFORM:

cstatus	Set to zero
comarealen	Set to total number of two-byte words in *comarea*.

VINITFORM may set the following *comarea* values:

cstatus	Set to non-zero value if call unsuccessful.
numerrs	Set to total number of fields in which errors were detected.
nfname	Set to new next form name, if name changed by processing specifications.
repeatapp	Set to new current form code, if code changed by processing specifications.
freezapp	Set to new next form code, if code changed by processing specifications.

Discussion

Certain values may be assigned to fields as initial values. These values are determined by special processing specifications that are explicitly or implicitly defined as part of the initialize phase of field processing using FORMSPEC. These values include any initial values specified on the Field Menus for the form. If no initial values were specified, all fields are initialized to blanks by VINITFORM. If the form being initialized is a child or sibling to the previous form, data from the previous form is transferred to this form (with conversion if necessary) before initializations occur. If a field in a child or sibling form must be initialized to blanks, use $EMPTY as an initial value.

VINITFORM only resets the field error flag if the field is initialized explicitly by initialization phase (INIT) processing specifications.

Example

COBOL

```
CALL "VINITFORM" USING COMAREA.
```

BASIC

```
140 CALL VINITFORM(Cl(*))
```

FORTRAN

```
CALL VINITFORM(COMAREA)
```

SPL/PASCAL

```
VINITFORM(COMAREA);
```

The calls shown above set initial values in the data buffer area of memory according to initialize specifications defined for each field in the current form.

VLOADFORMS

Loads specified forms into terminal local form storage memory.

Syntax

```
VLOADFORMS {comarea,numofforms,formsloaded,forms}
```

Parameters

comarea	Must be *comarea* name specified when the forms file was opened with VOPENFORMF. If not already set, the following *comarea* items must be set before calling VLOADFORMS:

cstatus	Set to zero.
language	Set to the code identifying the programming language of the calling program.
comarealen	Set to total number of two-byte words in *comarea*.

VLOADFORMS sets the following items in *comarea*:

cstatus	Set to nonzero value if call unsuccessful.

numofforms	Two-byte integer value, between 1 and 255, inclusive, indicating the number of forms to be loaded.
formsloaded	Two-byte integer value indicating the number of forms that were successfully loaded.
forms	The names of the forms to be loaded. Each name can be up to 15 characters and is stored in a 16-byte character array with a one byte filler that is not part of the name.

Discussion

This intrinsic is used on terminals having local form storage. VLOADFORMS loads the forms named in *forms* into terminal local form storage memory in the order they are specified. It may not be possible to load all the forms named because of a limit of available space in local form storage or in the form storage directory (set with *formstoresize* prior to VOPENTERM and VOPENFORMF). When there is no more space, any other *forms* named in *forms* are ignored. The value returned in the *formsloaded* parameter is:

formsadded + formsalreadypresent

where *formsadded* is the number of forms specified in the *forms* parameter that were *actually* loaded into the local storage of the terminal and *formsalreadypresent* is the number of forms specified in the *forms* parameter that were confirmed to *already* exist in the local storage. Because forms are loaded by form name, family relationships are ignored. Multiple family members can be loaded at the same time.

Note that the terminal keyboard may be locked briefly while VLOADFORMS verifies whether or not a form was loaded. In case keys are pressed during this time, the terminal beeps to indicate that the keystrokes and/or entered data are lost. You can avoid this by not calling VLOADFORMS between calls to VSHOWFORM and VREADFIELDS. Refer to the discussion on "Local Forms Storage" earlier in this section.

Example

COBOL

```
77 NUM-OF-FORMS         PIC S9(4)COMP.
77 FORMS-LOADED         PIC S9(4)COMP.
77 FORMS                PIC X(16).
    :
MOVE 1 TO NUM-OF-FORMS
MOVE "FORMA                "TO FORMS.
CALL "VLOADFORMS" USING COMAREA, NUM-OF-FORMS, FORMS-LOADED, FORMS.
```

BASIC

```
10 INTEGER N,F
15 DIM F$[16]
20 N=1
30 F$="FORMA       "
40 CALL VLOADFORMS(C[*],N,F,F$)
```

FORTRAN

```
INTEGER*2 NFORM,FLOAD
CHARACTER*16 FORMS
NFORM=1
FORMS="FORMA       "
CALL VLOADFORMS(COMAREA,NFORM,FLOAD,FORMS)
```

SPL/PASCAL

```
INTEGER
     NUM'OF'FORMS,
     FORMS'LOADED;
BYTE ARRAY
     FORMS(0:15);
      :
NUM'OF'FORMS:=1;
MOVE FORMS:="FORMA        ";
VLOADFORMS(COMAREA,NUM'OF'FORMS,FORMS'LOADED,FORMS);
```

VMERGE

VMERGE is a new VPlus utility that allows you to combine two or more separately-compiled VPlus forms files into a single forms file, which may then be used with an application program to manage the entry and/or retrieval of data.

Combining multiple forms files with VMERGE provides the following advantages:

- Some file size limitations can be overcome. Since there are limitations on how many physical records can be placed in a VFORM file, VPlus users sometimes find that they cannot use a single forms file to hold all the forms their application requires. With VMERGE, it is possible to add more forms (from a second forms file) to the ones in the initial forms file, even if the first is at or near the maximum size.
- It is sometimes easier to maintain a large forms file as several separately compiled "modules" that can then be combined with VMERGE rather than to maintain one large forms file.

Overview

VMERGE usually resides in PUB.SYS. It may be run on either MPE/iX or MPE V.

Before VMERGE is invoked, you must specify two input forms files and one output file. The input files are specified with file equations for the formal designators VMASTER and VAUX. The output file is specified with a file equation for the formal designator VOUTPUT. VMASTER and VAUX must exist, and each may be of type VFORM (slow forms file) or VFAST (fast forms file). VOUTPUT is created by the VMERGE utility and is of type VFAST.

As VMERGE runs, informative messages are presented on $STDLIST. If any problems are encountered, appropriate error messages are displayed. These message are described in the "VMERGE Messages" section in this article.

Forms File Handling by FORMSPEC and VMERGE

VPlus forms files exist in files with two different file codes: VFORM (slow forms file) and VFAST (fast forms file). VFORM files are created and modified with FORMSPEC. VFORM files contain the "source" for each form in the file, coded in a way that FORMSPEC can understand. When a forms file is compiled by FORMSPEC, "object" forms are added to the VFORM file. The object forms are accessed when the forms file is used with ENTRY or another application program that invokes the VPlus intrinsic functions.

When you use FORMSPEC to compile a VFORM file, you may request the creation of a fast forms file. This file contains only the object forms for the forms in the specified VFORM file. Processing the VFAST file is fast because the file is smaller than its corresponding VFORM file. That is, the fast forms file does not contain source forms and, therefore, can be accessed faster. Since a fast forms file does not contain source forms, it cannot be modified by FORMSPEC.

Both VFORM and VFAST files are limited in the number of records they can hold. However, the problem is more severe with VFORM files since they contain both source and object forms.

Previously, the only way to create a fast forms file was by compiling a VFORM file with FORMSPEC. Consequently, you might have been unable to include forms that theoretically could have fit into a VFAST file, since the source and object forms might have been too large to fit into a VFORM file. Now, VMERGE gives you an alternate method for generating VFAST files that contain additional forms. However, VFAST files still have limited capacity, and so there are still limits on the total number of forms you can place in a forms file, even using VMERGE.

Initial analysis shows that the ratio of fast form file size to slow form file size is around 1/3 to 1/10. This suggests that you can expect to combine forms from three or more nearly full (to FORMSPEC) forms files into one forms file, by using VMERGE. However, this is an estimate only, since non-typical forms files may vary considerably in their object to source ratios.

Input File Compatibility

Not all forms files can be successfully combined using VMERGE. The input forms files must be “compatible” in order to be combined with VMERGE. The compatibility factors are:

- Form names -- the same form name may not appear in both the VMASTER and the VAUX files.
- Save fields -- if both input files use save fields, the specifications for both files must be identical in all respects: names of save fields, lengths, data types, and initial values. The save fields must also be defined in the same order. It is permissible for one file to use save fields and the other not to do so.
- Global function key labels -- if both input files define global function key labels, the specifications for both files must be identical in all respects. It is permissible for one file to define global function key labels and the other not to do so. In this case, the global function key labels from the file which contains them will be retained in the output file.
- Terminal selection and language id -- both input files must have exactly the same set of terminals selected and the same Forms File Language specified (from FORMSPEC Terminal/Language Selection Menu).

There are a number of forms file characteristics that may differ between the two input files that are not serious enough for VMERGE to consider the two files as incompatible. These include: head form name, error enhancement, window display line, error window color, and window enhancement. The characteristic found in the VMASTER file is retained in the VOUTPUT file.

Application Requirement for Combined Forms Files

VMERGE takes two FORMSPEC-compiled forms files specified by the VMASTER and VAUX file designators, extracts the object forms from each file, and places these forms in a VFAST file specified by the VOUTPUT file designator.

Every forms file has a $HEAD form designated for it. Additionally, every form in a forms file has a “Next Form” designated for it. Next Form may be $HEAD or it may be the name of another form in the file. In order to compile a forms file, FORMSPEC requires that any form named as the $HEAD or as a Next Form exist in the file. Consequently, it is impossible for a form in the VMASTER forms file to refer to a form in the VAUX forms file as its Next Form and vice versa.

Therefore, the application program used with a combined forms file must be coded to sequence among the forms in the combined forms file without depending on the Next Form designation. The Next Form designation can only be used when it and the current form originated from the same forms file.

VMERGE users should be aware that VMERGE makes the $HEAD form from VMASTER the $HEAD for VOUTPUT. If data entry operators are used to seeing the $HEAD form from the VAUX file, they may be surprised if this $HEAD form is no longer what is displayed when they bring up their application.

Technical Reference

VMERGE normally resides in PUB.SYS; if it is moved from PUB.SYS, it must be moved to a group with "DS" capability.

VMERGE is invoked with the following command:

```
        :RUN VMERGE.PUB.SYS
```

Before invoking VMERGE, three file equations must be given. The two VMERGE input forms files are indicated by the file designators VMASTER and VAUX. VMASTER and VAUX must designate existing forms file, with file codes VFORM or VFAST. The VMASTER and VAUX files must have been compiled with a recent version of FORMSPEC. If an input file is provided that does not meet this criterion, a message is given, and VMERGE processing halts.

VMERGE's output VFAST forms files is indicated by the file designator VOUTPUT. If the designated file already exists and has the file code VFAST, then it is purged and recreated by VMERGE. If the file exists but has a file code other than VFAST, the file is not purged, a warning message is given, and VMERGE halts.

The files used by VMERGE are opened for exclusive use to avoid concurrent update problems.

Two JCWs (job control words) are defined for use with VMERGE: VMERGETERSE and VMERGEERROR. If the user sets VMERGETERSE to 1 before running VMERGE, then the messages indicating the form names contained in the VMASTER and VAUX files are suppressed. The VMERGEERROR JCW is set by VMERGE after it runs. If VMERGE detected a severe error that prevented the VOUTPUT file from being successfully created, VMERGEERROR is set to 1. If VMERGE successfully created the VOUTPUT file, VMERGEERROR is set to 0.

Example 1: Using VMERGE to Combine Forms Files Suppose your company has three distinct uses for an order form prepared by a salesperson. The order form is used in the shipping department, the billing department, and the marketing department. Currently the original order form is passed among three clerks who each enter their data using their own VPlus forms files (SHIPFF, BILLFF, and MARKETFF) that are distinct to their respective departments. Your job is to merge the forms files and fix the application program so that one clerk can enter the data for all three departments.

You decide that each of the three application areas (shipping, billing, and marketing) should be selectable by pressing a function key. Once an area has been selected, processing will be performed using the same sequence as when the application for that area existed on its own. Remember that while combining the three applications into one, you will need to examine the effect of $HEAD being different from what two of the original three programs expected. Most likely, you will want to create a new form to serve as the $HEAD for the combined application.

After examining and fixing the application program, you will need to look at the forms files and make them compatible, if they are not already. You will need to make sure that the save fields, if any, have distinct names and are identical in each of the forms files that uses them. Additionally, you will need to check that the global function key specifications are compatible in the three forms files. Note that in some cases incompatibilities may be so extensive that it will be impossible to merge the forms files.

Next you determine what order to merge the forms files together. Suppose in this example that the marketing forms file is much more volatile than the other two. Therefore, you decide that this file should be the last one merged. You also need to decide which form your combined application should use as its $HEAD. Suppose in this example you depend on the $HEAD form from BILLFF being the $HEAD of the combined forms file.

Combining the forms files is accomplished in two stages. In the first stage you enter:

```
:FILE VMASTER=BILLFF
:FILE VAUX=SHIPFF
:FILE VOUTPUT=INTERFF
:RUN VMERGE.PUB.SYS
```

VMERGE creates the intermediate forms file INTERFF.

In the second stage you enter:

```
:FILE VMASTER=INTERFF
:FILE VAUX=MARKETFF
:FILE VOUTPUT=COMBOFF
:RUN VMERGE.PUB.SYS
```

VMERGE creates the final forms file COMBOFF. You may wish to keep the intermediate forms file, INTERFF, so you won't need to recreate it if only MARKETFF changes.

Now you could test COMBOFF with your combined application program.

Example 2: Using VMERGE to Divide a Forms File Another use of VMERGE is to divide an existing forms file into two or more smaller forms files. This might be desirable if different forms in the original forms file were going to be modified by different people, or if compiling the entire forms file takes a long time.

When VMERGE is used to divide a file, there are few compatibility problems since the original form file is already "merged". You will have to be careful not to introduce incompatibilities (for example, refer to the $HEAD and Next Form issues discussed earlier) as a result of dividing the file.

Suppose you want to remove forms F1, F2, and F3 from a forms file named BIGFF and place them into another forms file, LITTLEFF. The F1, F2, and F3 forms are changed often, and you want to separate them out in order to minimize the time it takes to recompile each time changes are made.

First, you would FCOPY BIGFF to LITTLEFF by entering the following command:

```
:FCOPY FROM=BIGFF;TO=LITTLEFF;NEW
```

Second, you would run FORMSPEC on BIGFF to delete forms F1, F2, and F3. You would need to correct the Next Form fields for any affected forms.

Third, you would probably want to take advantage of some of FORMSPEC's batch mode facilities to delete all but the F1, F2, and F3 forms from LITTLEFF (see the Deleting Forms Using FORMSPEC's Batch Mode Facilities section below).

Fourth, you would compile both forms file to verify that no $HEAD or Next Form dependencies exist. The compilation for BIGFF will be lengthy, but it will only need to be done once.

Fifth, you would make your changes to the forms in LITTLEFF and add additional forms that could not be put into BIGFF due to size limitations.

After you had taken all these steps you could combine the two forms files with VMERGE:

```
:FILE VMASTER=BIGFF
:FILE VAUX=LITTLEFF
:FILE VOUTPUT=SUPERFF
:RUN VMERGE.PUB.SYS
```

Check your application using the SUPERFF forms file.

VOPENBATCH

Opens existing batch file for processing; or, if the specified file is new, creates a batch file and then opens it for processing.

Syntax

```
        VOPENBATCH {comarea,batchfile}
```

Parameters

comarea	Must be *comarea* name specified when the forms file was opened with VOPENFORMF. If not already set, the following comarea items must be set before calling VOPENBATCH:

cstatus	Set to zero.
language	Set to the code identifying the programming language of the calling program.
comarealen	Set to total number of two-byte words in *comarea*.

VOPENBATCH sets the following *comarea* items:

nfname	Set to the name of the form corresponding to the record identified by *recnum*.
recnum	Set to zero if new file opened; to the next sequential record number if existing file opened.
numrecs	Set to zero if new file opened; to the number of nondeleted records in file if existing file opened.

VOPENBATCH may set the following *comarea* items:

cstatus	Set to nonzero value if call unsuccessful.
filerrnum	Set to file error code if MPE file error.

batchfile	Character string of up to 36 characters (including a terminator) that identifies the batch file being opened. Specified name can be any fully qualified MPE file name.

Discussion

VOPENBATCH opens the specified batch file for processing by the calling program. The batch file may be an existing file or a new file.

Existing File

If the named file already exists, VOPENBATCH initializes *recnum* to the record number of the next record in the file, and *numrecs* to the total number of existing batch records. Thus, a user resuming collection does not overwrite data collected into previous batch records. VOPENBATCH sets *nfname* to the name of the form associated with the batch record to be collected. This record is identified by *recnum.* VOPENBATCH keeps track of forms sequence in order to associate a form with a record. For example, if a batch file is closed after record 6 of FORMA was collected, and FORMA is a repeating form, the batch file starts with record 7 FORMA when it is next opened by VOPENBATCH.

VOPENBATCH also resets the global environment (save fields and so forth) to the environment existing when collection stopped. (All this information is derived from the file labels preceding each file.)

New File

If the named file does not exist, VOPENBATCH creates a new file with the specified name and sets *recnum* and *numrecs* to zero. A new batch file created by VOPENBATCH has the following characteristics:

• Non-KSAM file	• Update access
• USASCII coded data	• Exclusive (nonshared) access
• Fixed-length records	• No dynamic locking
• No carriage control	• No multirecord access
• :FILE command allowed (use actual file designator)	• Normal buffering

Record Format

The size of the fixed-length batch file records must be large enough to hold the largest data buffer used by the forms file associated with the batch file, plus 10 two-byte words for batch record control information. If the largest data buffer is an even number of bytes, an additional two-byte word is added before the control information. This batch record information consists of:

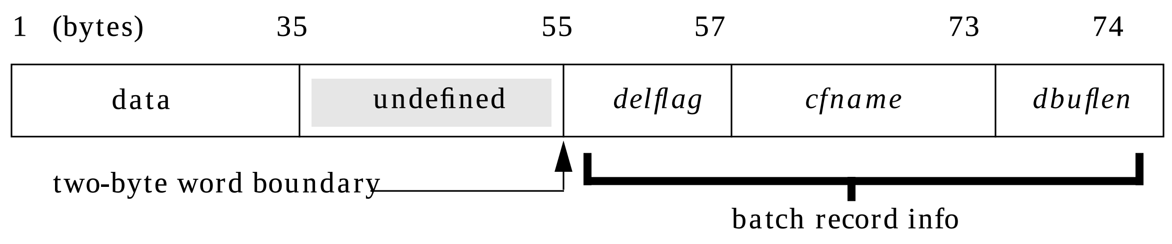
2 bytes (logical) — Delete flag (TRUE if record deleted)

16 bytes (character) — Current form name + extra character

2 bytes (logical) — Data buffer length

Total = 20 bytes

The above batch record information is written at the end of each record in the batch file, starting on a two-byte word boundary. To illustrate, assume the record size is 74 bytes, and the data only requires 35 bytes (bytes are numbered from 1):



Labels

In addition, VOPENBATCH creates sufficient user labels (each 256 bytes long) to hold any save field buffers, plus 176 bytes for the collection environment and the forms file version.

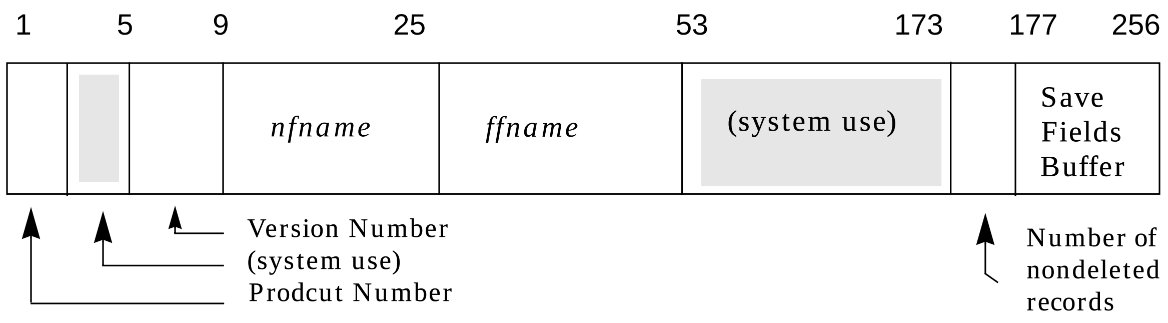
The length of the save field buffers depends on how many (if any) save fields were defined for the form and the length of each. The collection environment consists of the forms file name and version number, the next form name, and 122 bytes of system information. This information is stored as follows:

# of Bytes	
2	— Product number
2	— (System use)
4	— Forms file version (date and time of compilation)
16	— Next form name
28	— Forms file name
120	— Reserved for system use
4	— Number of nondeleted records in batch file

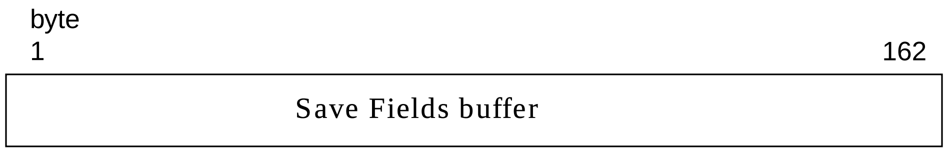
```
Total = 176
```

For example, assume that the save fields buffer requires 242 bytes. The following user labels are created by VOPENBATCH:

LABEL 0 — Contains first 80 bytes of Save Fields buffer



LABEL 1 — Contains remaining 162 bytes of Save Fields buffer

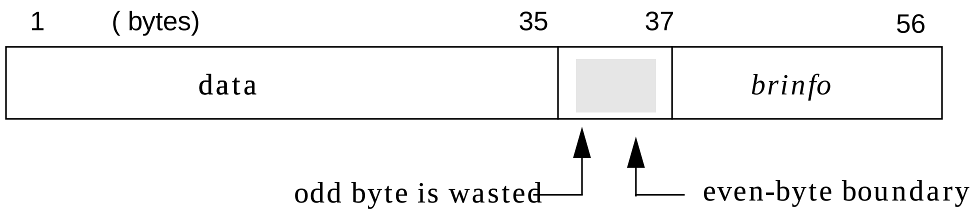


Note that the length of the save fields buffers is determined by taking the number of bytes in each save field, summing them together, adding one byte, and then, if odd, rounding the total up to an even number.

Creating Your Own Batch File

Should you want to create your own batch file rather than calling VOPENBATCH, you must build the file using the above record format and user label requirements. Also, if you create a batch file with variable-length or undefined-length records rather than fixed-length records, browsing of the batch file is not allowed. Undefined-length records are formatted like fixed-length records.

If you do, nevertheless, want to create a batch file with variable-length records for data collection only, the batch file information is stored immediately following the data in each record. Assume a variable-length record with 35 bytes of data:



The total size of this record is 56 bytes. Depending on the size of the data, other records have varying lengths. The maximum size of the variable-length records must be the size of the largest data buffer (in bytes) plus 20 bytes for the batch record information.

Example

COBOL

```
CALL "VOPENBATCH" USING COMAREA, BATCH.
```

BASIC

```
170 CALL VOPENBATCH(C(*),Bl$)
```

FORTRAN

```
CALL VOPENBATCH(COMAREA,BATCH)
```

SL/PASCAL

```
VOPENBATCH(COMAREA,BATCH);
```

If the requested batch file name is an existing file, it is opened and the user can continue to enter data into the record following the last record that contains data. If the requested batch file name is a new file, VOPENBATCH creates a batch file and data entered at the terminal is written to the first record in the file (record 0).

VOPENFORMF

Opens forms file for access.

Syntax

```
        VOPENFORMF {comarea,formfile}
```

Parameters

comarea The *comarea* name must be unique for each open forms file. If not already set, the following *comarea* items must be set before calling VOPENFORMF:

cstatus	Set to zero.
language	Set to the code identifying the programming language of the calling program.
comarealen	Set to total number of two-byte words in *comarea*.
usrbuflen	BASIC programs only; set to number of two-byte words needed for *comarea* extension.
labeloption	Set to zero or one.
formstoresize	Set to minus one, zero, or one through 255 to indicate the number of forms allowed in the form storage directory.

VOPENFORMF sets the following *comarea* items:

cstatus	Set to nonzero value if call unsuccessful.
lastkey	Set to zero.
numerrs	Set to zero.
recnum	Set to zero.
dbuflen	Set to zero.
cmode	Set to zero (collect mode).
repeatapp	Set to zero (no repeat/append).
freezapp	Set to zero (clear current form).
printfilnum	Set to zero.
deleteflag	Set to FALSE (all zeros) (used for batch file intrinsics).
cfname	Set to blank.
nfname	Set to name of head form.
cstatus	Set to nonzero value if call unsuccessful.
filerrnum	Set to file error code if MPE file error.
windowenh	Set to enhancement code defined in forms file.

If the data capture devices are used:

errorlight	Set to default or user specified value.

splitmsgpause	Set to default or user specified value.

formfile Character string of up to 36 characters (including a terminator) that identifies the forms file being opened. Specified name can be any fully qualified MPE file name.

Discussion

VOPENFORMF opens the specified forms file for processing by the calling application. If the calling application is BASIC, it must provide its own *comarea* extension immediately following the *comarea* and specify the size of this extension in *usrbuflen*. If the application is written in a programming language other than BASIC, a DL area is obtained for use as the *comarea* extension. In this case, the application must not use the DL area for other functions. Refer to Appendix E for more information on the DL area. The forms file listing gives an estimate of DL to DB area used. You use this estimate with the MAXDATA parameter at PREP or RUN time. The *labeloption* must be set to one if you are using function key labels.

Example

COBOL

```
CALL "VOPENFORMF" USING COMAREA, FNAME.
```

BASIC

```
100 CALL VOPENFORMF(c(*),F1$)
```

FORTRAN

```
CALL VOPENFORMF(COMAREA,FNAME)
```

SPL/PASCAL

```
BYTE ARRAY FNAME(0:34);
       :
MOVE FNAME:="FORMSA ";
VOPENFORMF(COM1,FNAME);
```

VOPENTERM

Opens a VPLUS supported terminal.

Syntax

```
        VOPENTERM {comarea,termfile}
```

Parameters

comarea The *comarea* name must be unique for each open forms file. The calling program should initialize the entire *comarea* to zero before calling VOPENTERM. For additional information about about using the pseudo intrinsic .LOC to put and address into a word of the COMAREA, refer to the *COBOL II/XL Reference Manual*.

In addition, the following *comarea* items must be set before calling VOPENTERM:

cstatus	Set to zero.
language	Set to the code identifying the programming language of the calling program.
comarealen	Set to total number of two-byte words of *comarea*.
formstoresize	Set to minus one, zero, or one through 255 to indicate the number of forms allowed in the form storage directory.

VOPENTERM may set the following *comarea* items:

cstatus	Set to nonzero value if call unsuccessful.
filerrnum	Set to file error code if MPE file error.
filen	Set to MPE file number of terminal.
identifier	Set to appropriate VPLUS terminal type code.
environ	Set to logical device number of terminal.
labinfo	Set to appropriate number and length of labels used by VPLUS.

If the data capture devices are used:

leftmodule	Set to appropriate value defining terminal.
rightmodule	Set to appropriate value defining terminal.
keyboard	Set to appropriate value defining terminal.
display	Set to appropriate value defining terminal.

termfile Character string of up to 36 characters (including a terminator) that identifies the terminal. If set to a blank, the session device is used to get the *ldev#* of the logon terminal. Otherwise, any fully qualified MPE file name can be assigned to the terminal as its formal file designator. If specified, the name must be terminated by a special character (a blank is suggested). Before using a terminal identified by name, the formal designator may be equated to an actual designator with a `:FILE` command.

Discussion

This intrinsic opens the terminal as a file. If you are running your program as a session with your terminal as the open terminal file, the terminal name should be left blank so that the session device is opened. If you are opening a form, you should call VOPENFORMF before using this intrinsic,

Once the terminal is opened for control by VPLUS, an application should not call system intrinsics to communicate with the terminal. All terminal I/O must be controlled through VPLUS intrinsics.

If you are using an HP 2640B or HP 2644 terminal and the terminal is not in block mode, VOPENTERM asks you to press the **BLOCK MODE** key; other terminals are set to block mode automatically. The data capture devices are treated as character mode terminals.

Example

COBOL

```
77 T1 PIC X(8) VALUE "VTERM
01 COMAREA.
     :
     :

PROCEDURE DIVISION.
     :
OPENTERMINAL.
     CALL "VOPENTERM" USING COMAREA, T1.
```

This example opens the device referenced by the file equation VTERM. If no file equation exists, the default is the logon device.

BASIC

```
90 Tl$-#" "
100 CALL VOPENTERM(C(*),T1$)
```

FORTRAN

```
T1=" "
VOPENTERM(COMAREA,T1)
```

SPL/PASCAL

```
MOVE T1:="          ";
VOPENTERM(COMAREA,T1);
```

The examples shown above open the session device with *comarea* identified as COMAREA (C(60) for BASIC).

VPLACECURSOR

Allows an application to position the cursor to any input field at run-time.

Syntax

```
        VPLACECURSOR {comarea,fieldnum}
```

Parameters

comarea Must be *comarea* name specified when the forms file was opened with VOPENFORMF. If not already set, the following *comarea* items must be set before calling VPLACECURSOR:

cstatus	Set to zero.
language	Set to the code identifying the programming language of the calling program.
comarealen	Set to total number of two-byte words in *comarea*.

VPLACECURSOR may set the following *comarea* items:

cstatus	Set to non-zero value if call unsuccessful.
filerrnum	Set to file error code if MPE file error.

fieldnum A two-byte integer that identifies the input field, where a positive number indicates the field number; a negative number indicates the screen order number.

Discussion

VPLACECURSOR allows an application to position the cursor to any input field at run-time. This intrinsic places the cursor to the input field specified by *fieldnum*. Calling VPLACECURSOR with a positive number indicates the field number; a negative number indicates the screen order number. Using the field number is preferable because if the fields in a form are rearranged, no modification to an application is necessary.

VPLACECURSOR must be called after calling VSHOWFORM. VPLACECURSOR returns an error if the field number or the screen order number does not exist. An error is also returned if the intrinsic is called to place the cursor to a display-only field.

Example

COBOL

```
    CALL "VPLACECURSOR" USING COMAREA,FIELD-NUM.
```

BASIC

```
    120 CALL VPLACECURSOR(C(*),FIELDNUM
```

FORTRAN

```
    CALL VPLACECURSOR(COMAREA,FIELDNUM)
```

SPL/PASCAL

```
    VPLACECURSOR(COMAREA,FIELDNUM);
```

VPOSTBATCH

Protects a user-specified portion of the batch file data from a system crash by posting an end-of-file mark after the last record referenced and updating the batch file labels.

Syntax

```
        VPOSTBATCH      {comarea}
```

Parameters

comarea	Must be *comarea* name specified when the forms file was opened with VOPENFORMF. If not already set, the following *comarea* items must be set before calling VPOSTBATCH:

cstatus	Set to zero.
comarealen	Set to total number of two-byte words in *comarea*.

VPOSTBATCH will set the following field:

cstatus	Set to nonzero if called unsuccessful, zero if successful.

Discussion

VPOSTBATCH posts an end-of-file mark after the last record referenced in the batch file and updates the environmental information found in the file label. Refer to the discussion of VOPENBATCH for a description of the environmental information.

If a system crash or power failure occurs while the batch file is open, all data before the end-of-file mark is preserved, and data collection continues from that point. In ENTRY, VPOSTBATCH is called after every 20 records, though you may extend or shorten this posting interval. Two cautions:

1. Never call VPOSTBATCH while you are in BROWSE mode, or at any time when the last record referenced is not the last record in the batch file. If you call this intrinsic when the last record referenced is in the middle of the file, VPOSTBATCH posts a mark before the actual end of the file, causing all data after this mark to be lost.
2. The *comarea* field *numrecs*, which contains the number of undeleted records in the file, may not be restored correctly after a system crash if batch records have been deleted since the last call to the VPOSTBATCH intrinsic.

Example

COBOL

```
CALL "VPOSTBATCH" USING COMAREA.
```

BASIC

```
290 CALL VPOSTBATCH(C(*))
```

FORTRAN

```
CALL VPOSTBATCH(COMAREA)
```

SPL/PASCAL

```
VPOSTBATCH(COMAREA);
```

VPRINTFORM

Prints the current form on an off-line list device.

Syntax

```
        VPRINTFORM {comarea,printcntl,pagecntl}
```

Parameters

comarea	Must be *comarea* name specified when the forms file was opened with VOPENFORMF. If not already set, the following *comarea* items must be set before calling VPRINTFORM:

cstatus	Set to zero.
comarealen	Set to total number of two-byte words in *comarea*.
printfilnum	Set to file number of the list file to which the form is to be printed. If set to zero, VPRINTFORM opens the device "LP" as the list file and sets *printfilnum* to the file number of the opened list file.

VPRINTFORM may set the following *comarea* values:

printfilnum	If VPRINTFORM opened the list file, set to the file number of the open file.
cstatus	Set to nonzero value if call unsuccessful.
filerrnum	Set to file error code if MPE file error.

printcntl	Two-byte integer that, if set to 1, causes VPRINTFORM to underline each field in the form listing. If set to any other value, fields are not underlined.
pagecntl	Two-byte integer value that determines the carriage control operation performed after a form is listed. May be any of the carriage control codes used by the MPE FWRITE intrinsic, including the following:

```
     61(octal)      Page eject
    320(octal)      No line feed or carriage return
   zero             Carriage return/line feed
```

Discussion

VPRINTFORM prints the current form to a list file. It is analogous to VSHOWFORM, in that it prints the form and the current data buffer values, except that VPRINTFORM prints the form on a hardcopy device rather than on the terminal. Enhancements obviously cannot be shown directly; the window line and key labels are not printed. The form must have been read into the form definition area of memory by a prior call to VGETNEXTFORM. The information printed depends on what is in the data buffer, which may not necessarily be the same as what is displayed.

The carriage control character specified in the *pagecntl* parameter is effective after the form is printed.

If the calling program opens the list file, it must supply the file number of this file in *printfilnum*. If *printfilnum* is zero, VPRINTFORM opens a list file and sets *printfilnum* to the file number of the file. VPRINTFORM opens the list file, with the formal and actual file designator FORMLIST, assigns it to the device class LP, and specifies its length as 80 bytes. This is equivalent to using the file equation:

```
:FILE FORMLIST;DEV=LP;REC=-80
```

A user may change any of these characteristics with an MPE :FILE command.

Example

COBOL

```
      CALL "VPRINTFORM" USING COMAREA, UNDERLINE, PAGE.
```

BASIC

```
      135 CALL VPRINTFORM(C(*),U,P)
```

FORTRAN

```
      CALL VPRINTFORM(COMAREA,UNDRLN,PAGE)
```

SPL/PASCAL

```
      VPRINTFORM(COMAREA,UNDERLINE,PAGE);
```

Each of the calls shown above prints the current form on a list device; if not already open, it opens the device file.

VPRINTSCREEN

Prints the entire contents of a terminal screen to an off-line list device or laser printer during VPLUS execution.

Syntax

```
VPRINTSCREEN {comarea, readsize}
```

Parameters

comarea If not already set, the following *comarea* items must be set before calling VPRINTSCREEN:

cstatus	Set to zero
comarealen	Set to the total number of two-byte words in *comarea*.
printfilenum	Set to the file number of the list file to which the form is printed. If set to 0, VPRINTSCREEN opens the device "LP" as the list file and sets *printfilenum* to the file number of the opened list file.

VPRINTSCREEN may set the following *comarea* items.

printfilenum	If VPRINTSCREEN opened the list file, set to the file number of the opened file.
cstatus	Set to a nonzero value if call is unsuccessful.
fileerrnum	Set to the MPE error code if an MPE file error is encountered.

readsize Two-byte integer, reserved for system use. Must be set to 0.

Discussion

This intrinsic provides the capability for printing the current screen display with function keys, line drawing characters and appended forms, as well as the data on the screen. It differs from VPRINTFORM, which is limited to printing only the data in the form data buffer. However, VPRINTFORM offers the option of underlining fields, which VPRINTSCREEN in LP mode does not.

The programmer can produce copies of VPLUS screens in either of two ways: incorporating VPRINTSCREEN into an application so that screen images can be captured at run-time with their data, or developing a simple utility that allows data to be entered into the screens before calling VPRINTSCREEN to reproduce them. The utility has the advantage of removing the overhead caused by VPRINTSCREEN from the application, while still providing a way to reproduce screens and data for product literature.

VPRINTSCREEN always uses the Pascal HEAP procedures to perform stack allocation. This introduces the risk of conflict for programs written in COBOL, FORTRAN/66 or SPL, because the existing intrinsics would use DLSIZE for stack allocation in these cases. Applications written in these languages must, therefore, follow two rules when calling VPRINTSCREEN. They are:

1. The language id in the *comarea* must be set to 5.

2. The INTRINSIC calling mechanism must be used when calling the VPLUS intrinsics from the main and all interacting parts of the application. For example, with COBOL the application must use:

```
    :CALL INTRINSIC <intrinsic name>
```

Refer to Appendix E of this manual, and to the COBOL and Pascal reference manuals, for more information on calling mechanisms.

For applications that use a language id of 5 in the VPLUS *comarea*, including Pascal, FORTRAN/77 and HPBUSINESS BASIC, VPLUS uses the HEAP procedures for stack allocation, so VPRINTSCREEN can be called in the standard format.

The programmer may implement this feature by defining a function key that allows the user to print the screen contents at any time. This would be useful for providing immediate output during production.

Modes of Operation

VPRINTSCREEN operates in two modes, normal and documentation mode. A VPLUS supported terminal is required for execution of this intrinsic.

Normal Mode

This is the default calling mode for VPRINTSCREEN. When called, the value in the *printnumfile* word of the *comarea* is used to determine the list device. If the calling program opens the list file, it must supply the file number of this file in *printfilenum* VPRINTSCREEN opens the list file with the formal and actual file designator FORMLIST, assigns it to the device class LP, and specifies its length as 80 characters. This is equivalent to using the file equation:

```
:FILE FORMLIST;DEV=LP;REC=-80
```

The user may change any of these characteristics with a :FILE command.

NOTE It is recommended that VPRINTSCREEN and VPRINTFORM not be used in the same program. Since the same list is used for both listings, output from the two calls will be intermixed.

Each time VPRINTSCREEN is called, a PAGE EJECT is performed at the end of the print operation.

Documentation Mode

You require TDP in order to use VPRINTSCREEN in this mode. VPRINTSCREEN, in conjunction with TDP, provides the capability to print screen contents on a laser printer (HP2680A or HP2688A). In this mode, field highlighting other than color, borders, alternate character sets and active function keys are captured and converted to the requisite font for printing on the laser printer.

Documentation mode is enabled by setting a JCW before running the program. It is:

```
:SETJCW VPRINTJCW=1
```

When VPRINTJCW is set to 1, the list file FORMLIST is NOT opened. Instead, a temporary file called EPOCLIST is created (or appended to, if it already exists). EPOCLIST can be saved and renamed on completion of the screen capture, then input to TDP and 'finaled'. The user can add text to the file or include it in a separate TDP file. Refer to the *TDP Reference Manual* for more information on use and include files.

The following files are sample files supplied on the FOS installation tape. They should be restored to a local group to be used by applications using VPRINTSCREEN in documentation mode. To accomplish this, mount the FOS tape and perform the following restore commands:

```
:file t;dev=TAPE:restore *t;V@.HP32209.HPPL89;local;show
```

These files may also be a non-local group, provided the application supplies a file equation, T. E.,

```
:file VSETUP=VSETUP.othergroup.otheraccount
```

```
Filename (in HP32209.HPPL89)    Description
---------------------------------------------------------------------
VENV80                          environment file for HP2680 printer
VENV88                          environment file for HP2688 printer
VSETUP                          TDP include file
VEPOCUSE                        TDP use file for merging screens
VPRTDEMO                        demo program for VPRINTSCREEN”
```

In order to print the forms on a laser printer, an environment file must be created (if it does not already exist). Refer to the *IFS/3000 Reference Guide* for more information. The environment files for the HP2680 and HP2688 laser printers are VENV80 and VENV88 respectively. All environment files must include the font ids listed under "Limitations" below.

EPOCLIST uses VSETUP, a TDP include file, as the default file to reference the environment files, which must be accessible to TDP in order for EPOCLIST to be printed. If an environment file other than VENV80 or VENV88 is used, EPOCLIST must be modified to reference this file, and the font definitions from VENV80 and VENV88 must be included in it (see "Limitations" below).

Printing Screens from TDP. To print the contents of EPOCLIST to a laser printer, follow these steps:

1. `:PURGE (or :RENAME) EPOCLIST`
2. `:SETJCW VPRINTJCW=1`
3. `:RUN <your application program>`
4. Use the print function wherever it is available to save the screen contents, including data, in EPOCLIST.
5. Exit the application.
6. `:SAVE EPOCLIST`
7. `:RENAME EPOCLIST, <new name>` If you do not rename EPOCLIST, the output generated the next time the application is run will be appended to the existing file.
8. `:RUN TDP.PUB.SYS`
9. Text in the file and execute the command, "Final from <new name> to *HP2680" (or

HP2688)

The screens will be printed out, one per page. Step 8 and 9 may be specified in a job stream.

Merging Screens with a TDP File. You can include screens in an existing TDP document easily by using the file VEPOCUSE, a TDP use file that divides EPOCLIST into separate files containing one screen per file. The VSETUP file must be included as one of the first statements of the TDP document so that the correct environment file is referenced for printing the screens.

Follow these steps:

1. :RUN TDP Clear the workspace.
2. :USE VEPOCUSE Answer the prompts, and enter a PREFIX. VEPOCUSE separates the old file into new files, each containing one screen.
3. Text in the TDP document and include each screen file in the correct place.

It is recommended that screen files be included as separate, individual files in a document, rather than being incorporated directly into the text, because EPOCLIST has a record size of 168 bytes and most document files are set to 80 bytes.

These files may also be a non-local group, provided the application supplies a file equation, T. E.,

```
:file VSETUP=VSETUP.othergroup.otheraccount
```

```
Filename (in HP32209.HPPL89)    Description
-------------------------------------------------------------------
VENV80                          environment file for HP2680 printer
VENV88                          environment file for HP2688 printer
VSETUP                          TDP include file
VEPOCUSE                        TDP use file for merging screens
VPRTDEMO                        demo program for VPRINTSCREEN”
```

Limitations

The following limitations pertain to the use of VPRINTSCREEN.

- Procedure calls must be modified in order to call the intrinsic from a language that does not use a VPLUS language id of 5 in the *comarea*, for example, COBOL, SPL and FORTRAN/66.
- It uses additional stack resources.
- Native Language support is NOT available for VPRINTSCREEN. To print a screen in another language, you require a LOCALIZED environment file which maps to the following fonts:

```
c = full bright
g = half-bright inverse video
d = normal
l = line-draw
m = math
```

- The current environment files do not distinguish between full bright and half bright, but the code is set up to do so.

- Screen images cannot be scaled; fonts come in one size only.
- The maximum TDP record size is 168 characters. A single line in a screen may easily exceed this limit if it contains may escape sequences, in which case the line may be truncated.
- TDP macros 5-9 are used to minimize truncation, but some screens will reach the limit anyway. If you use your own macros 5-9, some inconsistencies may appear when text and screens ar merged.
- TDP may indicate that errors have occurred when, in fact, there are none. The most common error messages are: "Unrecognizable command" and "n characters have been truncated".

Example

```
This is a smaple EPOCLIST file.

*>>DATE: FRI, MAY 3, 1987, 11:19AM

if main in hpvsetup.pub
image 28
need 28
>>>>>>screen formatting commands<<<<<<
*>>END VPRINTSCREEN B.05.00
```

VPUTBUFFER

Copies data from an application to the data buffer in memory.

Syntax

```
     VPUTBUFFER {comarea,buffer,buflen}
```

Parameters

comarea	Must be *comarea* name specified when the forms file was opened with VOPENFORMF. If not already set, the following *comarea* items must be set before calling VPUTBUFFER:

cstatus	Set to zero.
language	Set to the code identifying the programming language of the calling program.
comarealen	Set to total number of two-byte words in *comarea*. Must be at least 70 words in length if the ARB feature is used.
buffercontrol	Set bit 15 to 1 to indicate that data is to be transformed according to the ARB specifications.

VPUTBUFFER may set the following *comarea* items:

cstatus	Set to nonzero value if call unsuccessful
numerrs	May be decremented as a result of new data replacing data in field with error.
bufferstatus	Bit 15 set to 1 if data conversion is successful

buffer	Character string in a application containing the data to be copied to the data buffer in memory.
buflen	Two-byte integer variable that specifies the number of bytes to be copied to the data buffer in memory. The *comarea* item *dbuflen*, which contains the size of the data buffer in memory, may be used as this parameter.

Discussion

This intrinsic transfers data from a buffer in an application to the data buffer in memory. The length of the data moved is based on the number of bytes specified in the *buflen* parameter and the number of bytes in the largest data buffer in the forms file, whichever is less. The length of the buffer assigned to the current form is not considered since the user may intend the data for another form with a longer buffer length.

For example, assume there are three forms in the forms file:

- Form A *dbuflen* = 100 bytes
- Form B *dbuflen* = 200 bytes
- Form C *dbuflen* = 75 bytes

In this case, the maximum data buffer length is 200 bytes. If the current form is form A and the user calls VPUTBUFFER with a user buffer length (*buflen* parameter) of 200, he may intend to call VGETNEXTFORM to get form B and then VSHOWFORM to display form B with the 200 bytes of data moved to the data buffer with VPUTBUFFER.

Fewer bytes than the data buffer can hold may be transferred; the remaining space in the data buffer is not changed.

The data moved to the data buffer is exactly as it appears in the application buffer. (If you want the data converted to USASCII in the data buffer, you must use VPUTtype, where type is the data type of the field in an application.) When the data is displayed, it is moved to each field in the form in sequence from left to right, top to bottom. If any field being replaced by user data contained an error, VPUTBUFFER clears the error flag for the field and decrements *numerrs*.

Special Considerations

Designers using the ARB feature in VPUTBUFFER should be aware that damaging run-time errors could occur if the application is inadvertently run on a system that has a VPLUS version earlier than B.05.00.

To prevent this, the designer should do three things:

1. Document the product with a clear warning requiring VPLUS B.05.00 or later.
2. Use the VPLUS intrinsic HP32209 in the code. This intrinsic checks to make sure you are using the proper VPLUS version. If not, the application should terminate with an appropriate message.
3. Check offset 65 (*bufferstatus*) in the *comarea* on return from VPUTBUFFER. Bit 15 will be set to 1 if VPUTBUFFER performed the conversion successfully. In other words, the application must check both *status* and *bufferstatus* to be sure that the data was correctly converted.

Example

COBOL

```
01 DAT1.
     03 FIRSTNAME PIC X(6).
     03 LASTNAME PIC X(18).
        .
        .
        .
ACCEPT DAT1.
CALL "VPUTBUFFER" USING COMAREA, DAT1, DBUFLEN.
```

BASIC

```
235 L1=24
240 CALL VPUTBUFFER(C(*),D1$,L1)
```

FORTRAN

```
CALL VPUTBUFFER(COMAREA,DAT1,DBLEN)
```

SPL/PASCAL

```
BYTE ARRAY DAT1(0:23);
VPUTBUFFER(COM1,DAT1,LEN);
```

The following calls transfer 24 bytes from an application area, DAT 1 to the data buffer. In this example, the longest *dbuflen* is assumed to be 80 bytes.

VPUTFIELD

Copies data from an application into a specified field in the data buffer in memory.

Syntax

```
        VPUTFIELD {comarea,fieldnum,fieldbuf,buflen,actualen,nextfldnum}
```

Parameters

comarea	Must be *comarea* name specified when the forms file was opened with VOPENFORMF. If not already set, the following *comarea* items must be set before calling VPUTFIELD:

cstatus	Set to zero.
language	Set to the code identifying the programming language of the calling program.
comarealen	Set to total number of two-byte words in *comarea*.

VPUTFIELD may set the following *comarea* items:

cstatus	Set to nonzero value if call unsuccessful.
numerrs	May be decremented if new value is moved to a field which has error flag set.

fieldnum	Two-byte integer variable containing the number assigned to the field in the data buffer by FORMSPEC.
fieldbuf	Character string in an application containing the data to be copied to the specified data buffer field.
buflen	Two-byte integer containing the number of bytes to be copied from *fieldbuf* to the field identified by *fieldnum*.
actualen	Two-byte integer that specifies the number of bytes actually copied.
nextfldnum	Two-byte integer to which VPUTFIELD returns the number of the next field in screen order. If there are no more fields, it returns zero.

Discussion

The data in an application is copied to the field in the data buffer identified by its field number. Note that the field number is a unique number assigned to the field by FORMSPEC when the form is first created. The number assigned to a field by FORMSPEC does not change regardless of any changes to the field's position in the form or to its length. The field number can only be changed with the batch command, RENUMBER, as described in Section 7. The field number must not be confused with the screen order number, which is the field's position in the data buffer and corresponds to its position in the form.

If the field is shorter than the data transferred to it, the data is truncated on the right. If the field is longer than the data transferred to it, the data, if any, in the remaining space in the field is not changed.

If the field whose data is being replaced contained an error, VPUTFIELD clears the field's error flag, and decrements *numerrs*.

Note that VPUTFIELD does not convert the data. To convert data from internal numeric representation to character strings, you must use VPUTtype, where type specifies the data type of the field in an application.

Example

COBOL

```
MOVE 1 TO FIELDNUM.
MOVE 10 TO FIELD-LEN.
MOVE "GADGET        "TO PART-DES.
CALL "VPUTFIELD" USING COMAREA, FIELDNUM, PART-DES, FIELD-LEN,
                              DESC-LEN, NEXT-FIELD.
```

BASIC

```
250 F1=1
255 C=10
257 I$="GADGET   "
260 CALL VPUTFIELD(C(*),F1,I$,C,A,N)
```

FORTRAN

```
FLDNUM=1
ICOUNT=10
XITEM="GADGET          "
CALL VPUTFIELD(COMAREA,FLDNUM,XITEM,ICOUNT,INUM,FLDNXT)
```

SPL/PASCAL

```
INTEGER
   FLD'NUM,
   COUNT,
   ACTUAL'LEN,
   NXT'FLD'NUM;
   BYTE ARRAY PART'DES(0:9):="GADGET          ";
      :
   FLD'NUM:=1;
   COUNT:=10;
   VPUTFIELD(COMAREA,FLD'NUM,PART'DES,COUNT,ACTUAL'LEN,NXT'FLD'NUM);
```

The calls shown above copy a 10-byte value from an application to field number 1 in the data buffer.

VPUTSAVEFIELD

Copies data from an application to the specified save field in memory.

Syntax

VPUTSAVEFIELD {*comarea,sfname,sfbuf,buflen,actualen*}

Parameters

comarea	Must be *comarea* name specified when the forms file was opened with VOPENFORMF. If not already set, the following *comarea* items must be set before calling VPUTSAVEFIELD:

cstatus	set to zero.
language	set to the code identifying the programming language of the calling program.
comarealen	set to total number of two-byte words in *comarea*. Must be at least 70 words in length.

VPUTSAVEFIELD may set the following *comarea* item:

cstatus	set to nonzero value if call is unsuccessful.

sfname	a character string in the application which contains the name of the save field.
sfbuf	a character string in the application whose contents are copied to the save field in memory.
buflen	a two-byte integer variable which specifies the number of bytes in *sfbuf*.
actualen	a two-byte integer variable which specifies the number of bytes actually moved to the save field in memory.

Discussion

VPUTSAVEFIELD transfers the contents of the application variable specified by *sfbuf* to the save field in memory specified by *sfname*.

If the length of the save field is less than the length specified by *buflen*, the right most bytes are truncated. Following a successful transfer, *actualen* is set to the actual number of bytes transferred to the save field in memory.

Note that the contents of the save field as recorded in the forms file is not changed by a call to VPUTSAVEFIELD. Also VPUTSAVEFIELD does not convert the data moved to the save field in memory.

Example

```
COBOL
CALL "VPUTSAVEFIELD" USING COMAREA SFNAME SFBUF BUFLEN ACTLEN.

SPL
VPUTSAVEFIELD(COMAREA,SFNAME,SFBUF,BUFLEN,ACTLEN);
```

These calls will transfer the contents of the application buffer SFBUF to the save field in memory specified by SFNAME. If the call is successful ACTLEN will contain the exact number of bytes transferred.

VPUTtype

Copies a numeric value of specified type from an application to a field in the data buffer in memory, converting the value to character set coded external representation.

Syntax

```
        VPUTtype {comarea,fieldnum,variable}
```

Parameters

type The *type* in VPUTtype indicates that this intrinsic may be specified as:

VPUTINT	converts value to two-byte integer
VPUTDINT	converts value to four-byte integer
VPUTREAL	converts value to four-byte real value
VPUTLONG	converts value to eight-byte long value
*VPUTPACKED	converts packed decimal format to character set coded ASCII; this intrinsic has two extra parameters, *numdigits* and *decplaces*. Both are two-byte integer variables that contain the number of digits and number of decimal places, respectively, specified by the COBOL *usage is comp-3* data declaration.
*VPUTZONED	converts zoned decimal format to character set coded ASCII; has the parameters *numdigits* and *decplaces*, which are two-byte integer variables that contain the number of digits and number of decimal places, respectively, specified by the COBOL data declaration.
VPUTYYMMDD	converts six-byte character value to YMD, DMY or MYD

For example:

```
CALL "VPUTINT" USING COMAREA,FIELDNUM,VARIABLE
```

comarea Must be *comarea* name specified when the forms file was opened with VOPENFORMF. If not already set, the following *comarea* items must be set before calling VPUTtype:

cstatus	Set to zero.
comarealen	Set to total number of two-byte words in *comarea*.

VPUTtype may set the following *comarea* items:

cstatus	Set to nonzero value if call unsuccessful.
numerrs	May be decremented if new value replaces the value of a field with an error.

fieldnum Two-byte integer variable containing the field number assigned by FORMSPEC to the field in the data buffer to which the value is copied.

variable Variable of specified type in an application that contains the value to be converted to USASCII and copied to a field in the data buffer.

Discussion

Depending on how it is specified, this intrinsic converts integer, double, real, long, packed, zoned or yymmdd values to the external representation and copies the converted value to a particular field in the data buffer, right justified. (Note that the exact format depends on the data type of the destination field.) The destination field is identified by the field number assigned by FORMSPEC. The field to which the value is copied may be defined as a numeric field (data type NUM[n], IMPn, or DIG) or as a character field (data type CHAR).

NOTE If you are using VPUTBUFFER in conjunction with an ARB, you do not need to use VPUTtype. VPUTBUFFER performs all the required conversions on the application data in the buffer.

You can use both VPUTBUFFER with an ARB and VPUTtype calls in the same program: the *buffercontrol* setting in the *comarea* that controls ARB processing can be switched on or off for each form.

Note that the field number never changes as long as the field exists. It is not changed when the position of the field in the form is changed, or its length or other characteristics are changed. The field number can only be changed with the batch command, RENUMBER, as described in Section 7. The field number should not be confused the screen order number, which is the position of the field in the data buffer and is based on the field position within the form. Thus, the field number provides a way to locate fields regardless of their position.

If the specified field had an error, VPUTtype clears the field's error flag, and decrements *numerrs*.

Refer to Table 6-13. under VGETtype for the format of each of the data types that may be converted. Note that COBOL does not have type real or long, and BASIC does not have a double integer data type.

Example

COBOL

```
77 FIELD-NUM PIC S9(4) COMP.
77 ITEM PIC S9(4) COMP.
     :
MOVE 4 TO FIELD-NUM.
MOVE 25 TO ITEM.
CALL "VPUTINT" USING COMAREA, FIELD-NUM, ITEM.
```

BASIC

```
260 F1=4
263 I=25
265 CALL VPUTINT(C(*),F1,I)
```

FORTRAN

```
FIELD=4
ITEM=25
CALL VPUTINT(COMAREA,FIELD,ITEM)
```

```
SPL/PASCAL
INTEGER FIELD,ITEM;
   :
FIELD:=4;
ITEM:=25;
VPUTINT(COMAREA,FIELD,ITEM);
```

The calls shown above convert an integer value of 25 in the application to the external representation and copy it to field 4 in the data buffer in memory.

VPUTWINDOW

Copies a message to the window area of memory.

Syntax

```
VPUTWINDOW {comarea,message,length}
```

Parameters

comarea	Must be *comarea* name specified when the forms file was opened with VOPENFORMF. If not already set, the following *comarea* items must be set before calling VPUTWINDOW:
	language — Set to the code identifying the programming language of the calling program.
	comarealen — Set to total number of two-byte words in *comarea*.
	VPUTWINDOW may set the following *comarea* items:
	cstatus — Set to nonzero value if call unsuccessful.
message	Character string containing the message to be copied to the window area of memory.
length	Two-byte integer that specifies the number of bytes in the message. If set to zero, any message in the window is cleared to blanks. The maximum length is 150 bytes, but only 79 or fewer can be printable characters.

Discussion

This intrinsic copies the specified message to the window area of memory for later display. Then, a call to VSHOWFORM can be used to display the message in the window area of the terminal screen, with the window enhanced as indicated by *windowenh* of *comarea*. A message copied by VPUTWINDOW overwrites any previous message in the window area, including any message copied by a previous call to VPUTWINDOW or VSETERROR.

If the message is longer than the defined window length, the message is truncated on the right. If shorter, the rest of the window line is cleared.

Note that the forms file may be defined with no window line for error and status messages. In this case, the message is ignored.

Example

COBOL

```
MOVE "ENTER ORDERS ON THIS FORM" TO MESSAGE.
MOVE 25 TO MSG-LENGTH.
CALL "VPUTWINDOW" USING COMAREA, MESSAGE, MSG-LENGTH.
```

BASIC

```
310 M1$="ENTER ORDERS ON THIS FORM"
320 L1=25
330 CALL VPUTWINDOW(C(*),M1$,L1)
```

FORTRAN

```
MSG="ENTER ORDERS ON THIS FORM"
LEN=25
CALL VPUTWINDOW(COMAREA,MSG,LEN)
```

SPL/PASCAL

```
BYTE ARRAY MSG(0:24):="ENTER ORDERS ON THIS FORM";
INTEGER LEN;
 :
LEN:=25;
VPUTWINDOW(COMAREA,MSG,LEN);

The calls shown above copy the message

ENTER ORDERS ON THIS FORM

to the window area of memory.
```

VPUTYYYYMMDD

Converts a numeric value representing a date from an application and copies the converted value to a field in the data buffer in memory.

Syntax

```
VPUTYYYYMMDD {comarea,fieldnum,variable}
```

Parameters

comarea	must be *comarea* name specified when the forms file was opened with VOPENFORMF. If not already set, the following *comarea* items must be set before calling VPUTYYYYMMDD:

cstatus	set to zero.
comarealen	set to total number of two-byte words in *comarea*. Must be at least 70 words in length.

VGETYYYYMMDD may set the following *comarea* item:

cstatus	set to nonzero value if call is unsuccessful or requested field has an error.
numerrs	may be decremented if the new value replaces the value of a field with an error.

fieldnum	is a two-byte integer variable which specifies the number assigned to the field by FORMSPEC.
variable	is a character string in the application which contains the value to be converted.

Discussion

VPUTYYYYMMDD converts the contents of the application variable to the date order of the field specified by *fieldnum* and copies the converted value into the corresponding field in the data buffer, right justified. The application variable should contain a numeric value in YYYYMMDD format. The YYYYMMDD format is an 8-byte ASCII value, for example, 19961225. If errors occur during conversion, *cstatus* is set to an error code.

Refer to the VPUT*type* intrinsic description in the *Data Entry and Forms Management System VPLUS Reference Manual* for related information.

Example

```
COBOL
CALL "VPUTYYYYMMDD" USING COMAREA FIELDNUM VARIABLE.

SPL
VPUTYYYYMMDD(COMAREA,FIELDNUM,VARIABLE);
```

These calls will convert the contents of VARIABLE to a date value and place the converted value in the data buffer corresponding to the field specified by FIELDNUM.

VREADBATCH

Reads contents of current batch record into data buffer in memory.

Syntax

```
        VREADBATCH {comarea}
```

Parameters

comarea	Must be *comarea* name used when the batch file was opened with VOPENBATCH. If not already set, the following *comarea* items must be set before calling VREADBATCH:

cstatus	Set to zero.
comarealen	Set to total number of two-byte words in *comarea*.
recnum	Set to the number of the record in the open batch file from which data is to be read (records are numbered from zero).

VREADBATCH may set the following *comarea* items:

nfname	Set to the name of the form associated with the data read from the batch file (used by VGETNEXTFORM to retrieve this form from forms file).
dbuflen	Set to length of data buffer (in bytes) based on length of data read from batch record.
deleteflag	Set to TRUE (all ones) if delete flag in batch record indicates record is deleted; set to FALSE (all zeros) otherwise.
cstatus	Set to nonzero value if call unsuccessful.
filerrnum	Set to file error code if MPE file error.

Discussion

Use of this intrinsic is demonstrated by the browse/modify mode of ENTRY. It enables a user to view the data in the batch file. VREADBATCH may also be used to bring the data from a batch file into the data buffer so that it can be retrieved by an application with the VGETBUFFER, VGETFIELD, or VGET*type* intrinsics.

To display the data just read onto its associated form, VGETNEXTFORM must be called before the call to VSHOWFORM.

VREADBATCH reads the record (*recnum*) in the batch file opened by VOPENBATCH. The record is read into memory, where VREADBATCH extracts the batch record control information (refer to the VOPENBATCH description). This information includes the current form name which is moved to *nfname*, the delete flag which is moved to *deleteflag*, and the data length in bytes which is moved to *dbuflen*. The data buffer is not updated if the *deleteflag* has been set.

In order to use VREADBATCH, the batch file must be on a direct-access device and must be created with fixed-length records, *not* variable-length records.

Example

COBOL

```
CALL "VREADBATCH" USING COMAREA.
```

BASIC

```
175 CALL VREADBATCH(C(*))
```

FORTRAN

```
CALL VREADBATCH(COMAREA)

SPL/PASCAL
VREADBATCH (COMAREA);
```

The calls shown above read the batch record specified by *recnurm* update the *comarea* according to the batch record information stored with the data, and put the data in the data buffer in memory.

VREADFIELDS

Accepts all user input from an open terminal, including data entered by pressing **ENTER**, or user requests made by pressing a function key. Allows look-ahead form loading.

Syntax

```
        VREADFIELDS {comarea}
```

Parameters

comarea	Must be *comarea* named specified when terminal file was opened with VOPENTERM. If not already set, the following *comarea* items must be set before calling VREADFIELDS:

cstatus	Set to zero.
comarealen	Set to total number of two-byte words in *comarea*.

The following *comarea* items can be set:

lookahead	Set to zero to enable look-ahead form loading or to one to disable look-ahead form loading.
showcontrol	Set to control the touch feature or to affect the display on data capture devices. The particular settings are described below.

VREADFIELDS may set the following *comarea* values:

numerrs	Set to zero.
lastkey	Set to code for terminal function key pressed by user.
cstatus	Set to nonzero value if call unsuccessful.

Discussion

VREADFIELDS accepts user-entered data when **ENTER** is pressed. It maps the data into the data buffer in memory. The data is mapped in screen order, from left to right, top to bottom. If there are any display-only fields already in the buffer, the fields read by VREADFIELDS are interspersed among the display-only fields according to the screen order. VREADFIELDS then sets *lastkey* to zero. Note that the keyboard is locked after the data is read (the subsequent call to VSHOWFORM unlocks the keyboard). If a function key is pressed, VREADFIELDS sets *lastkey* to the corresponding number: 1 for f1, 2 for f2, and so forth. The screen is *not* read and data is not transferred when a function key is pressed. It is, however, possible to accomplish this by enabling the AUTOREAD feature. To enable AUTOREAD, a bit must be set in the *termoptions* position of the *comarea*. For further information regarding the AUTOREAD feature, consult the Terminal Communications Area section of Appendix G.

Application programs must supply the code to perform each of the functions that can be requested via a function key. The particular function assigned to a key is determined only by how the program processes the key code passed to it by VREADFIELDS (refer to Appendix A for examples of applications using VREADFIELDS).

Normally, there is no time limit for VREADFIELDS, whether the intrinsic is to be terminated by the enter key or by a function key. Timeouts can be enabled in an application by setting bits in the *termoptions* word of the *comarea* and specifying a timeout interval in the *usertime* word of the *comarea*. For more information regarding user timeouts, consult the Terminal Communications Area section of Appendix G.

For any terminal with touch capability, an application activates the touch feature by setting bit 0 of *showcontrol* to one. Then, for each time the user touches a field, VREADFIELDS returns *lastkey* with the field number of the touched field (as a negative number). Refer to Appendix G for the "Supported Terminal Features" list.

For the data capture devices, VREADFIELDS, not VSHOWFORM, displays a form one field at a time and reads entered data, also one field at a time. Editing is performed after data for all the fields in the form has been read. VREADFIELDS sets *lastkey* as follows for these terminals: -1 is the **ATTENTION** key, 0 is the **ENTER** key, 1 is the "A" key, 2 is the "B" key…and 26 is the key. If a function key is pressed when entering data on the data capture devices, VREADFIELDS returns to the calling program and does *not* display any remaining fields.

Bits 12 and 11 of the *comarea* item *showcontrol* affect form display on the data capture devices. These bits can be set as follows:

bit 12 = 0	Stop after displaying a form without fields.
1	Do not stop after displaying a form without fields.
11 = 0	Do not put a right closing bracket (]) on all input fields.
1	Put a right closing bracket (]) on all input fields.

A form without fields can be used to display a message to a user. If bit 12 is zero, this type of form is frozen until **ENTER** or a function key is pressed so that the user has time to read the message. If bit 11 is one, a right bracket is displayed as a terminator on all input fields.

VREADFIELDS may also invoke look-ahead form loading for terminals with local form storage using the *comarea* item *lookahead*. If point-to-point data communications is being used, the next form (as defined in the forms file) is loaded into the terminal's local form storage memory while the user is typing in data) before **ENTER** is pressed. If multipoint or X.25 data communications is being used, the next form is loaded after the data has been read, after **ENTER** is pressed. If there is no room in local form storage for the next form, or if there are no available entries in the form storage directory (set with *formstoresize* prior to VOPENTERM and VOPENFORMF), the least recently used form could be purged from local form storage. Look-ahead form loading does not load multiple family members. If a family member of the next form is already in local form storage, this family member is changed into the required form when it is displayed. (Refer to Section 4 for more information on Local Form Storage.)

If the next form does not load successfully, no error is returned. The next form is simply displayed by VSHOWFORM.

Example

COBOL

```
     CALL "VREADFIELDS" USING COMAREA.
```

BASIC

```
130 CALL VREADFIELDS(C(*))
```

FORTRAN

```
CALL VREADFIELDS(COMAREA)
```

SPL/PASCAL

```
VREADFIELDS(COMAREA);
```

Each of the following calls accepts user input from the terminal, transfers any data to the data buffer, and sets *lastkey*.

VSETERROR

Sets the error flag of a specified field and increments *numerrs*. If this is the first field (in screen order) with an error, it copies a message to the window area of memory for later display.

Syntax

```
VSETERROR {comarea,fieldnum,message,msglen}
```

Parameters

comarea	Must be *comarea* name specified when the forms file was opened with VOPENFORMF. If not already set, the following *comarea* items must be set before calling VSETERROR:	
	cstatus	Set to zero.
	language	Set to the code identifying the programming language of the calling program.
	comarealen	Set to total number of two-byte words in *comarea*.
	VSETERROR may set the following *comarea* items:	
	numerrs	Contains number of fields in form with errors; incremented by VSETERROR.
	cstatus	Set to nonzero value if call unsuccessful.
fieldnum	Two-byte integer variable containing the field number or screen order number of the data field to be flagged for error. If it is the screen order number, it must be a negative number representing the position of the field on the form.	
message	Character string containing an error message to be copied to the window area of memory for subsequent display by VSHOWFORM.	
length	Two-byte integer containing the length in bytes of the message parameter. If length is set to -1, the current content of the window is not changed. If length is set to zero, the current content of the window is cleared to blanks.	

Discussion

This intrinsic can be called by any program that wants to perform its own edits, either in addition to, or instead of VFIELDEDITS. VSETERROR sets the error flag associated with the specified field. If this is the first time this field has been diagnosed as having an error, VSETERROR increments *numerrs*; otherwise, it does not change *numerrs*. Thus, if a field has an error detected by a prior call to VFIELDEDITS, a call to VSETERROR for that field does not increment *numerrs*. However, if the touch feature is enabled, VSETERROR *toggles* the error flag for a specified field to clear, if it was already set, and decrements *numerrs*. Refer to "Coding the Touch Feature" in Appendix E.

If this is the first field in the form (in screen order) that has an error, the specified message is copied to the window area of memory for later display. If you do not want to change the current contents of the window, set the length parameter to -1. To clear a message, set length to zero.

Note that the field number identifies a field regardless of its position in the form. Thus, field "1" could be the third field in screen order counting from left to right, top to bottom. Using the field order number is preferable because if the fields in a form are rearranged, no modification to an application is necessary.

Example

COBOL

```
      DATA DIVISION.
          77 FLDNUM         PIC 9(4) COMP.
          77 MESSAGE        PIC X(80).
          77 MLENGTH        PIC S9(4) COMP.
               :
      PROCEDURE DIVISION.
          MOVE 3 TO FIELDNUM.
          MOVE 22 TO MLENGTH.
          MOVE "THIS FIELD IS REQUIRED" TO MESSAGE.
          CALL "VSETERROR" USING COMAREA, FLDNUM, MESSAGE, MLENGTH.
```

BASIC

```
          220 F1=3
          225 L1=22
          230 M$="THIS FIELD IS REQUIRED"
          250 CALL VSETERROR(C(*),F1,M$,L1)
```

FORTRAN

```
          FF=3
          ML=22
          MSG="THIS FIELD IS REQUIRED"
          CALL VSETERROR(COMAREA,FF,MSG,ML)
```

SPL/PASCAL

```
          INTEGER FF,ML;
          BYTE ARRAY MESSAGE(0:21);="THIS FIELD IS REQUIRED";
            :
          FF:=3;
          ML:=22;
          VSETERROR(COMAREA,FF,MESSAGE,ML);
```

The examples above set error flags for field number "3" in the currently open form, and set up the message "THIS FIELD IS REQUIRED" to be displayed if no value is entered in the field and this is the first field (in screen order) in which an error is detected.

The following examples show how VSETERROR can be used to set error flags for a field in error without writing a message to the window.

COBOL

```
   MOVE 3 TO FIELDNUM.
   MOVE -1 TO MLENGTH.
   CALL "VSETERROR" USING COM1, FIELDNUM, MESSAGE, MLENGTH.
```

BASIC

```
   220 F1=3
```

```
230 L1=-1
CALL VSETERROR(C(*),F1,M$,L1)
```

FORTRAN

```
FF=3
ML=-1
CALL VSETERROR(COM1,FF,MSG,ML)
```

SPL/PASCAL

```
FIELD:=3;
LEN:=-1;
VSETERROR(COM1,FIELD,MESSAGE,LEN);
```

VSETKEYLABEL

Allows for temporarily setting, programmatically, a new label for a function key.

Syntax

```
        VSETKEYLABEL {comarea,formorglob,keynum,label}
```

Parameters

comarea Must be *comarea* name specified when the forms file was opened with VOPENFORMF. If not already set, the following *comarea* items must be set before calling VSETKEYLABEL:

cstatus	Set to zero.
language	Set to the code identifying the programming language of the calling program.
comarealen	Set to total number of two-byte words in *comarea*.

VSETKEYLABEL may set the following *comarea* items:

cstatus	Set to nonzero value if call is unsuccessful.

formorglob Integer specifying which type of label is to be temporarily set.

- 0 - Set global label.
- 1 - Set current form label.

keynum Integer from 1 to 8 corresponding to function key to be set.

label A byte array containing the text for the label; must be 16 bytes long.

Discussion

VSETKEYLABEL is only a temporary setting of a new label for an individual function key. Use of this intrinsic does not change the label definition made in FORMSPEC. Note only one function key can be set with this intrinsic. The *labeloption* must be set to one prior to VOPENFORMF.

The temporary label is displayed after the next call to VSHOWFORM. If the temporary label is global, it remains active until the forms file is closed or it is replaced by a new global label. If the temporary label is for the current form only, it is replaced when the next form is retrieved or when a new current form label is set.

If no global or current form labels have been defined with FORMSPEC or no labels have been set with VSETKEYLABELS, the key label buffers are cleared before the label being defined with this intrinsic is set.

Example

COBOL

```
     77 FORM-OR-GLOB    PIC S9(4)COMP.
     77 KEY-NUM         PIC S9(4)COMP.
     77 KEY-LABEL       PIC X(16).
         :
```

```
MOVE 1 TO FORM-OR-GLOB.
MOVE 1 TO KEY-NUM.
MOVE "LABEL      1    " TO KEY-LABEL.
CALL "VSETKEYLABEL" USING COMAREA, FORM-OR-GLOB, KEY-NUM, KEY-LABEL.
```

BASIC

```
10 INTEGER F,N
20 DIM L$[16]
30 F=1
40 N=2
50 L$="LABEL       1   "
60 CALL VSETKEYLABEL(C[*],F,N,L$)
```

FORTRAN

```
INTEGER FORMORGLOB,KEYNUM
CHARACTER*16 LABEL
FORMORGLOB=1
LABEL="LABEL       1   "
CALL VSETKEYLABEL(COMAREA,FORMORGLOB,KEYNUM,LABEL)
```

SPL/PASCAL

```
INTEGER
    FORM'OR'GLOB,
    KEY'NUM;
BYTE ARRAY
    KEY'LABEL(0:15);
     :
FORM'OR'GLOB:=1;
KEY'NUM:=1;
MOVE KEY'LABEL:="LABEL       1      ";
VSETKEYLABEL(COMAREA,FORM'OR'GLOB,KEY'NUM,KEY'LABEL);
```

VSETKEYLABELS

Allows for temporarily setting, programmatically, labels for function keys.

Syntax

```
        VSETKEYLABELS {comarea,formrglob,numoflabels,labels}
```

Parameters

comarea	Must be *comarea* name specified when the forms file was opened with VOPENFORMF. If not already set, the following *comarea* items must be set before calling VSETKEYLABELS:

cstatus	Set to zero.
language	Set to the code identifying the programming language of the calling program.
comarealen	Set to total number of two-byte words in *comarea*.

VSETKEYLABELS may set the following *comarea* items:

cstatus	Set to nonzero value if call is unsuccessful.

formorglob	Integer specifying which type of labels are to be temporarily replaced.

- `0` - Replace global labels.
- `1` - Replace current form labels.

numoflabels	Integer from 0 through 8 indicating how many labels are to be temporarily set, where 0 indicates that the labels defined in FORMSPEC should be used.
labels	A byte array in which the labels are defined. The length of the array must be at least *numoflabels* * 16.

Discussion

VSETKEYLABELS is only a temporary setting of new labels for the function keys. Use of it does not change the label definitions made in FORMSPEC. The *labeloption* must be set to one prior to VOPENFORMF.

The temporary labels are displayed after the next call to VSHOWFORM. If the temporary labels are global, they remain active until the forms file is closed or replaced by new global labels. If the temporary labels are current form labels, they are replaced when the next form is retrieved or when new current form labels are set.

If no global or current form labels have been defined with FORMSPEC or no labels have been set with a prior call to VSETKEYLABELS, the key label buffers are cleared before the labels currently being defined are set.

Example

COBOL

```
     77 FORM-OR-GLOB      PIC S9(4)COMP.
     77 NUM-OF-LABELS     PIC S9(4)COMP.
```

```
77 KEY-LABELS       PIC X(32).
   :
MOVE 1 TO FORM-OR-GLOB.
MOVE 2 TO NUM-OF-LABELS.
MOVE "LABEL       1     LABEL     2     " TO KEY-LABELS.
CALL "VSETKEYLABELS" USING COMAREA,FORM-OR-GLOB,NUM-OF-LABELS,KEY-LABELS.
```

BASIC

```
10 INTEGER F,N
20 DIM L$[32]
30 F=1
40 N=2
50 L$="LABEL       1  LABEL      2      "
60 CALL VSETKEYLABELS(C[*],F,N,L$)
```

FORTRAN

```
INTEGER FORMORGLOB,NUMLABELS
CHARACTER*32 LABELS
FORMORGLOB=1
NUMLABELS=2
LABELS="LABEL      1      LABEL      2       "
CALL VSETKEYLABELS(COMAREA,FORMORGLOB,NUMLABELS,LABELS)
```

SPL/PASCAL

```
INTEGER
  FORM'OR'GLOB,
  NUM'OF'LABELS;
BYTE ARRAY
  LABELS(0:31);
   :
FORM'OR'GLOB:=1;
NUM'OF'LABELS:=2;
MOVE LABELS:="LABEL      1     LABEL    2      ";
VSETKEYLABELS(COMAREA'FORM'OR'GLOB,NUM'OF'LABELS,LABELS);
```

VSETLANG

The VSETLANG intrinsic specifies the native language to be used with an international forms file. For more information on Native Language Support, see Section 8.

Syntax

```
    VSETLANG {comarea,langnum,errorcode}
```

Parameters

comarea	Must be *comarea* name specified when the forms file was opened with VOPENFORMF. If not already set, the following comarea items must be set before calling VSETLANG:
	comarealen Set to the total number of two-byte words in *comarea*.
	VSETLANG may set the following *comarea* items:
	cstatus Set to nonzero value if call is unsuccessful.
langnum	A two-byte integer, *passed by value*, containing the native language ID number of the forms file language to be used by VPLUS.
errorcode	Two-byte integer to which the error code is returned.

Discussion

This intrinsic sets the native language to be used by VPLUS at run time for an international forms file. The forms file must be opened before calling VSETLANG. Otherwise, *cstatus* returns a nonzero value.

If VSETLANG is called to set the native language ID for a language-dependent or unlocalized forms file, an error code of -1 will be returned to *errorcode*. For international forms files, both *cstatus* and *errorcode* return a value of zero and the forms file is processed with the native language ID specified in *langnum*.

Example

COBOL

```
    CALL "VSETLANG" USING COMAREA,\LANGNUM\,ERRORCODE
```

BASIC Pass by value parameters not supported.

FORTRAN

```
    CALL VSETLANG (COMAREA,LANGNUM,ERRORCODE)
```

SPL/PASCAL

```
    VSETLANG(COMAREA,LANGNUM,ERRORCODE)
```

VSHOWFORM

Displays on the terminal screen the current form from local form storage or from the form definition buffer, any data in the data buffer, and any messages from the window buffer. Displays any labels from the key label buffer.

Syntax

```
VSHOWFORM {comarea}
```

Parameters

comarea Must be the *comarea* name specified when the terminal file was opened with VOPENTERM. If not already set, the following *comarea* items must be set before calling VSHOWFORM:

cstatus	Set to zero
comarealen	Set to total number of two-byte words in *comarea*.
windowenh	Set to window enhancement code; may be set before call to code for nondefault enhancement; otherwise, set by VOPENFORMF to default enhancement specified in forms file. If set to zero, window is not enhanced. (Refer to *windowenh* discussion under "Communication Area" earlier in this section.)

The following *comarea* item can be set:

showcontrol	Set to change VSHOWFORM control options. The particular settings are described below.

VSHOWFORM may set the following *comarea* values:

cstatus	Set to nonzero value if call unsuccessful.
filerrnum	Set to file error code if MPE file error.

(Note that *showcontrol* is not cleared by VSHOWFORM.)

Discussion

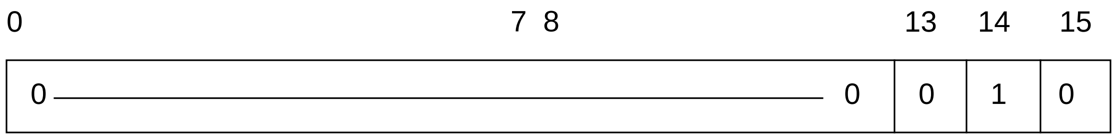
VSHOWFORM displays, on an open terminal screen, the form currently stored as a form image in the terminal's local form storage or in the form definition area of memory. Any enhancements specified for the form are used for the display. Data currently in the data buffer in memory is moved to the appropriate fields of the displayed form. Any message in the window buffer of memory is displayed in the line of the form selected as the status line. Also, any current form or global function key labels in the key label buffer are displayed.

Performance Optimization

In order to optimize VSHOWFORM performance, only changed information is written to the terminal. Thus, if the window has not been changed by VPUTWINDOW or VSETERROR since the last execution of VSHOWFORM, or if the current form is being repeated in place, these areas are not rewritten. Also, when a form is repeating in place, only the changed values in the data buffer area of memory are written by VSHOWFORM. This will be sufficient for most applications; however, these three optimizations (form, data, window) can be overridden by setting the *comarea* item *showcontrol* as follows:

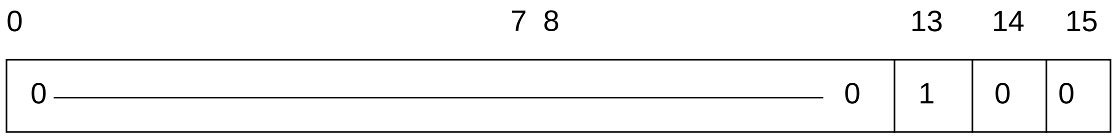
`bit 15 = 1`	Force form to be written to the terminal screen.
`14 = 1`	Force data and field enhancements to be written to the terminal screen.
`13 = 1`	Force window line to be written to the terminal screen.

Depending on the bits set, VSHOWFORM writes a form or data or the window to the terminal whether or not it has changed. Anything that has changed is always written to the terminal regardless of *showcontrol*. Any combination of these bits may be set. For example, if you want to force a write of all data:



showcontrol = octal 2

Or, if you want to force the window to be written:



showcontrol = octal 4

Display and Local Form Storage

Prior to display on terminals with local form storage, the form can be loaded into the terminal with any of the following methods:

- by a call to VLOADFORMS,
- by the previous call to VREADFIELDS if look-ahead form loading is enabled, or
- by VSHOWFORM if preload is enabled.

A prior call to VGETNEXTFORM reads the form into the form definition area of memory prior to display by VSHOWFORM. The form definition includes the form image, any field editing specifications and all enhancements.

The field enhancements are specified with the form definition. If a field has an error, VSHOWFORM changes its enhancement to the error enhancement defined for the form by FORMSPEC.

The data buffer may contain data as a result of initialization by VINITFORM, retrieval of user-entered data by VREADFIELDS, data formatting or movement caused by editing specified with each field and executed by VINITFORM, VFIELDEDITS, or VFINISHFORM. Data may also be transferred directly to the data buffer, either from an application with VPUTBUFFER or VPUTtype, or from a batch file with VREADBATCH.

The window contains any message set by VSETERROR or VPUTWINDOW.

Controlling the Keyboard

As soon as the form is displayed, VSHOWFORM normally enables the keyboard so the user can enter data. The next call is usually to VREADFIELDS, which locks the keyboard after the entered data is read. In case of consecutive calls to VSHOWFORM, the following *showcontrol* bit can be set:

`bit 10 = 0`	Enable the keyboard.
`1`	Do not enable the keyboard.

The scenario is as follows:

- the last VREADFIELDS call locked the keyboard,
- for each of a series of consecutive calls to VSHOWFORM; set bit 10 of *showcontrol* to one so the keyboard is not enabled,
- for the last of the consecutive calls to VSHOWFORM, set bit 10 to zero to enable the keyboard.

This ensures that keystrokes do not change a form while it is being displayed. Consecutive calls to VSHOWFORM could be used to display a form with no fields, append another form to it, and display this second form without doing a read in between.

Controlling Preload of Forms

On terminals with local form storage, VSHOWFORM first determines if the current form is already in the terminal. If the form is in local storage, it is displayed from local storage. If the form is not in local storage, it is preloaded into the terminal from the form definition area of memory depending on *showcontrol*, which can be set as follows:

`bit 9 = 0`	Do not preload the form.
`1`	Preload the form.

If bit 9 is zero, the form is written directly from the form definition area of memory to the terminal screen. If bit 9 is one, the form is preloaded into local storage and then displayed from local storage. One or more forms could be purged from local storage if there is not room for the form that is being loaded. Note that purging could occur even when *lookahead* is set to one.

Controlling the Touch Feature

On terminals with the touch feature, touch can be enabled or disabled with *showcontrol*, which can be set as follows:

`bit 0 = 0`	Do not enable the touch feature.
`1`	Enable the touch feature.

Controlling the Bell

By default (*termoptions* bit 15 is set to zero), VSHOWFORM sounds the bell if neither the screen nor the window line have been redisplayed. The bell can be suppressed in all cases by setting the *termoptions* bit to one.

Example

COBOL

```
CALL "VSHOWFORM" USING COMAREA.
```

BASIC

```
120 CALL VSHOWFORM(C(*))
```

FORTRAN

```
CALL VSHOWFORM(COMAREA)
```

SPL/PASCAL

```
VSHOWFORM (COMAREA);
```

The calls shown above display a form with optional data and enhancements on the terminal screen opened with the *comarea*, COM 1.

VTURNOFF

The VTURNOFF intrinsic turns off VPLUS block mode and enables character mode access without disturbing the terminal screen. The syntax and parameter descriptions for this intrinsic are provided below.

Syntax

```
VTURNOFF {COMAREA}
```

Parameters

COMAREA — Must be COMAREA name specified when the forms file was opened with VOPENTERM. If not already set, the following COMAREA items must be set before calling VTURNOFF:

CSTATUS	Set to zero.
COMAREALEN	Set to total number of 2-byte words in COMAREA.

VTURNOFF may set the following COMAREA fields:

CSTATUS	Set to nonzero value if call is unsuccessful or if field's error flag is set.
FILERRNUM	Set to file error code if MPE file error.

VTURNOFF is used for momentarily switching from VPLUS block mode to character mode. This procedure is designed for use after a terminal has been previously opened by VOPENTERM or after a VTURNON.VTURNOFF reconfigures the terminal and driver for character mode operations without disturbing the screen image on the terminal. The following operations normally performed in VCLOSETERM are not performed in VTURNOFF:

- Clear local form storage
- Enable the USER/SYSTEM keys
- Disable touch reporting, delete touch fields
- Clear screen
- Unlock keyboard
- Close terminal file

Note that VTURNOFF does not close the terminal file. To close the file and completely reset the driver and the terminal, VCLOSETERM must be used.

The following examples illustrate a call to VTURNOFF:

COBOL:

```
CALL “VTURNOFF” USING COMAREA.
```

BASIC:

```
200 CALL VTURNOFF(C(*))
```

FORTRAN:

```
CALL VTURNOFF(COMAREA)
```

SPL:

```
VTURNOFF(COMAREA);
```

PASCAL:

```
VTURNOFF(COMAREA);
```

VTURNON

The VTURNON intrinsic turns on VPLUS block mode without disturbing the terminal screen. The syntax and parameter descriptions for this intrinsic are provided below.

Syntax

```
    VTURNON {COMAREA,TERMFILE}
```

Parameters

COMAREA — The COMAREA name must be unique for each open forms file. The COMAREA must be the same COMAREA used in VOPENTERM. The following COMAREA items must be set before the call, if not already set:

CSTATUS	Set to zero.
LANGUAGE	Set to code that identifies the programming language of the calling program.
COMAREALEN	Set to total number of 2-byte words in COMAREA.

VTURNON may set the following COMAREA fields:

CSTATUS	Set to nonzero value if call is unsuccessful or if field's error flag is set.
FILERRNUM	Set to file error code if MPE file error.
FILEN	Set to MPE file number of terminal.
IDENTIFIER	Set to appropriate VPLUS/V terminal ID.
LAB'INFO	Set to appropriate number and length of labels.

TERMFILE — Must be the same terminal file name used in VOPENTERM.

VTURNON is normally used in an application when the terminal is already opened by VOPENTERM, and VTURNOFF was called to switch out of VPLUS block mode. VTURNON switches the application back to VPLUS block mode without disturbing the image on the terminal screen.

VTURNON reconfigures the terminal and the driver without performing the following operations which are normally performed by VOPENTERM:

- Initialize local form storage
- Clear screen
- Enable the USER function keys
- Disable or enable the USER/SYSTEM key, as specified in the SHOWCONTROL word

Unlike VOPENTERM, VTURNON will not ask you to press the BLOCK MODE key, if you are using an HP 2640B or HP 2644 terminal when the terminal is not in block mode.

The following examples illustrate a call to VTURNON using common programming languages:

COBOL:

```
CALL “VTURNON” USING COMAREA, T1.
```

BASIC:

```
90 T1$=” “
100 CALL VTURNON(C(*),T1$)
```

FORTRAN:

```
T1=” “
VTURNON(COMAREA,T1);
```

SPL:

```
MOVE T1:=”  “;
VTURNON(COMAREA,T1);
```

Pascal:

```
T1:=‘ ’;
VUTURNON(COMAREA,T1);
```

VUNLOADFORM

Unloads the specified form from local form storage memory.

Syntax

```
        VUNLOADFORM {comarea,whichform}
```

Parameters

comarea Must be *comarea* name specified when the forms file was opened with VOPENFORMF. If not already set, the following *comarea* items must be set before calling VUNLOADFORM:

cstatus	Set to zero.
language	Set to the code identifying the programming language of the calling program.
comarealen	Set to total number of two-byte words in *comarea*.

VUNLOADFORM may set the following *comarea* values:

cstatus	Set to nonzero value if call unsuccessful.

whichform The name of the form to be removed from local form storage. Each name can be up to 15 characters and is stored in a 16-byte character array with a one byte filler that is not part of the name.

Discussion

This intrinsic is used on terminals having local form storage. VUNLOADFORM purges the form named by the *whichform* parameter from terminal local form storage memory. Note that the terminal keyboard may be locked briefly while VUNLOADFORM verifies whether or not the form was purged. In case keys are pressed during this time, the terminal beeps to indicate that the keystrokes and/or entered data are lost. You can avoid this by not calling VUNLOADFORM between calls to VSHOWFORM and VREADFIELDS.

Example

COBOL

```
     77 WHICH-FORM       PIC X(16).
        :
     MOVE "FORMA       " TO WHICH-FORM.
     CALL "VUNLOADFORM" USING COMAREA, WHICH-FORM.
```

BASIC

```
      10 DIM W$[16]
      20 W$="FORMA         "
     100 CALL VUNLOADFROM(C[*],W$)
```

FORTRAN

```
     CHARACTER*16 WFORM
     WFROM="FORMA          "
     CALL VUNLOADFORM(COMAREA,WFORM)
```

SPL/PASCAL

```
     BYTE ARRAY WHICH'FORM(0:15);
       :
     WHICH'FORM:="FORMA             ";
     VUNLOADFORM(COMAREA,WHICH'FORM);
```

VWRITEBATCH

Writes a record to the batch file from the data buffer in memory, or deletes a record from the batch file

Syntax

```
        VWRITEBATCH {comarea}
```

Parameters

comarea	Must be *comarea* name specified when the batch file was opened with VOPENBATCH. If not already set, the following *comarea* items must be set before calling VWRITEBATCH:

cstatus	Set to zero.
comarealen	Set to total number of two-byte words in *comarea*.
recnum	Set to record number in batch file to which data is to be written (record numbers start with zero). The *recnum* is initialized to zero by VOPENBATCH (or to the next sequential record number if the file is an existing file).
deleteflag	Set to TRUE (all ones) if record is to be deleted.

(The *cfname* and *dbuflen* should be set by a prior call to VGETNEXTFORM.)

VWRITEBATCH may set the following *comarea* values:

cstatus	Set to nonzero value if call unsuccessful.
filerrnum	Set to file error code if MPE file error.
numrecs	Incremented each time a new record is written; decremented if record is deleted.

Discussion VWRITEBATCH writes the contents of the data buffer to the record specified by *recnum* in an open batch file. (The *recnum* must be maintained by the calling program.) VWRITEBATCH writes the following information to the batch record:

- Contents of the data buffer
- Batch record control information (from *comarea*):

deleteflag	TRUE if record is deleted.
cfname	Name of the form associated with this data.
dbuflen	Length (in characters) of the data buffer.

(Refer to "Record Format" in the VOPENBATCH description for a diagram of the batch record.)

VWRITEBATCH may be used in both the data collection and modify modes.

Data Collection Mode The data in the data buffer is entered on a particular form displayed at a terminal and then read into the data buffer by VREADFIELDS.

VWRITEBATCH can then be called to write the data buffer and the record control information to the batch record specified by *recnum*.

Modify Mode When data is modified, an existing batch file record is rewritten. The calling program must set *recnum* to the record number of this record.

To mark a batch record as deleted, the *deleteflag* must be set to TRUE (all ones) by an application. Then a call to VWRITEBATCH sets a corresponding flag in the batch record to mark the record as deleted. Since a deleted record still exists in the batch file, it can be viewed through FCOPY or a user-written intrinsic.

In ENTRY, a function key pressed by the user not only determines which record is to be viewed, but also specifies which record is to be deleted.

Example COBOL

```
CALL "VWRITEBATCH" USING COMAREA.
```

BASIC

```
165 CALL VWRITEBATCH(C(*))
```

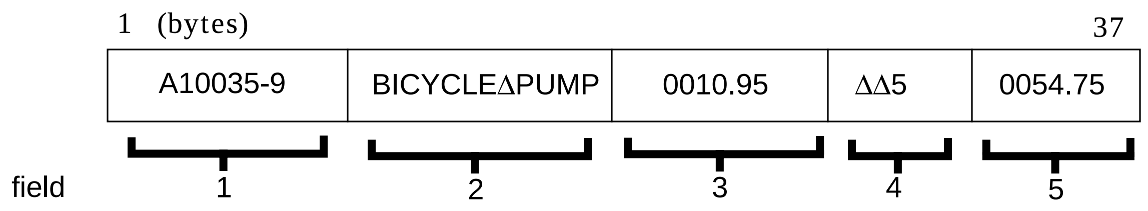
FORTRAN

```
CALL VWRITEBATCH(COMAREA)
```

SPL/PASCAL

```
VWRITEBATCH(COMAREA);
```

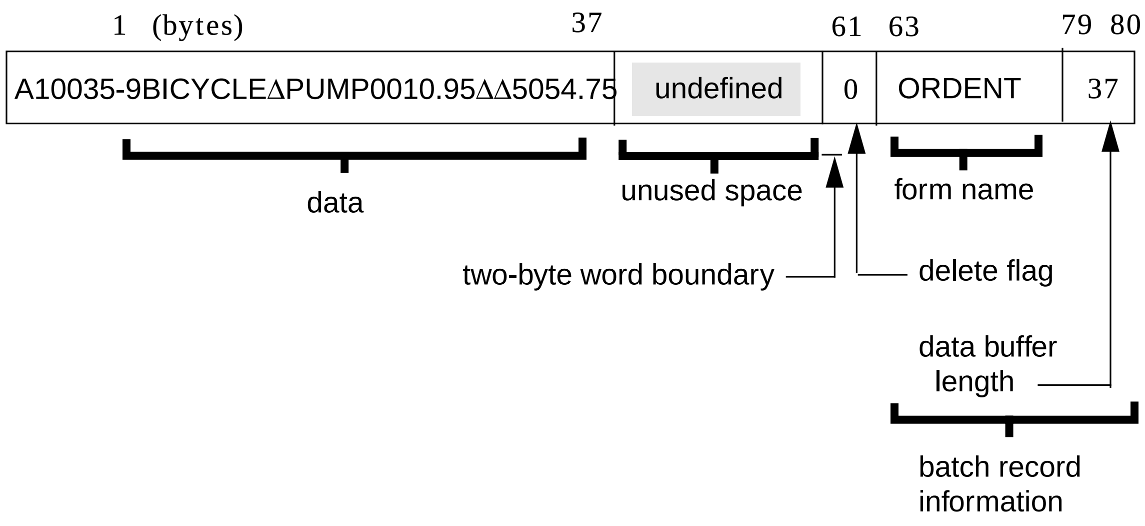
The calls shown above write the contents of the data buffer to the batch record identified by *recnum*. Assume the following data is in the data buffer and that it was entered on form ORDENT. (Note that the data entered on separate fields of a form is concatenated in the data buffer, with no separators.)



Assume the *comarea* is set as follows:

- *recnum* = 5
- *cfname* = ORDENT
- *dbuflen* = 37

In addition, assume the batch file opened by `VOPENBATCH` has fixed-length records, 80 bytes long. The call to `VWRITEBATCH` writes the following record as the sixth record in the batch file:



7 USING FORMSPEC IN BATCH MODE

USING FORMSPEC IN BATCH MODE

FORMSPEC in batch mode allows you to manage a forms file without tying up your terminal. With this feature, forms may be deleted from or copied to a forms file, compiled, and listed on a printer by accepting commands from a job stream or from a disc file or I/O device.

Deleting Forms Using FORMSPEC's Batch Mode Facilities

Suppose you wanted to delete all forms but F1, F2, and F3 from a forms file named LITTLEFF. To accomplish this, you would take the following steps:

1. Use the FORMSPEC "FORMS" command to generate a list of forms in LITTLEFF and redirect the list to a disk file.

   ```
   :FILE FORMOUT;REC=-80,16,F,ASCII;DEV=DISC;TEMP
   :RUN FORMSPEC.PUB.SYS;INFO="$STDIN"
   >FILE LITTLEFF
   >FORMS
   >EXIT
   :SAVE FORMOUT
   :RENAME FORMOUT,FORMCMDS
   :RESET FORMOUT
   ```

2. Bring FORMCMDS into an editor. Delete all lines that do not list form names. Delete the lines listing forms F1, F2, and F3. Insert the keyword "DELETE" in front of every other form name. Delete the text remaining after the form name.
3. Add the command `FILE LITTLEFF` as the first line in the file.
4. Add an EXIT command as the last line in the file.
5. Keep the update FORMCMDS file and exit the editor.
6. Execute the commands in the FORMCMDS file. Note: you may want to do this within a job file.

Compiling Forms File in Batch Mode

Compiling a forms file can be very time consuming, especially if there are many forms, fields, and field specifications defined in the file. By using FORMSPEC in batch mode, long compilations can be executed from a job stream. You do not have to wait for compilation to complete in order to use your terminal for other purposes.

Updating Forms In Batch Mode

A forms file can be updated systematically by using FORMSPEC in batch mode. Entire forms can be copied to or deleted from a forms file, field attributes may be updated, and function key labels may be created and updated. Individual screens, save fields, and some global information; however, cannot be defined in batch mode. Also, save fields cannot be copied from one forms file to another.

Listing Forms Files in Batch Mode

By using the batch mode LIST command, forms can be listed on the list device from a streamed job. This means forms file listings may be automated in a job that updates a forms file. For example, a job that updates a forms file could also print all or specific forms for future reference. The form listings may also be directed to a disc file to be examined.

Relating Forms In Batch Mode

Form family relationships can be established using the RELATE command in batch mode. The RELATE command requires that the forms exist and their screen designs be identical. Furthermore, the forms to be related must be distinct forms and the proposed child form cannot already be a member of a family of forms.

INVOKING FORMSPEC IN BATCH MODE

FORMSPEC in batch mode can be invoked from within a job stream or from within a session. When invoked, FORMSPEC can accept batch mode commands from a command file or from $STDIN.

To invoke FORMSPEC to accept batch mode commands from a command file, use the MPE :RUN command with the INFO=*filename* parameter, where *filename* is the name of the command file. For example:

```
:RUN FORMSPEC.PUB.SYS;INFO="COMMANDS.PUB.MFG"
```

will execute FORMSPEC in batch mode using COMMANDS.PUB.MFG as the command file. Remember, before you can invoke FORMSPEC in batch mode to accept commands from a command file, you must first create the command file using a text editor and enter the batch mode commands into the file. (See the end of this section for a sample of a command file.)

The INFO=*filename* parameter directly invokes batch mode. In addition, FORMSPEC will always execute in batch mode within a job stream. When the INFO=*filename* parameter is omitted within a job stream, $STDIN becomes the command file by default and commands are accepted from within the job stream. In order to use $STDIN as the command file within a session, the INFO=*filename* parameter is required. For example:

```
:RUN FORMSPEC.PUB.SYS;INFO="$STDIN"
```

In response to the greater than (>) prompt, enter the batch mode commands. Although this usage of batch mode will not free up a terminal, users will have the ability to react to any error conditions which may occur.

When FORMSPEC detects an error in batch mode, the error message is sent to the output file ($STDLIST by default). The number of errors can be checked with FORMSPECERRORJCW; number of warnings can be checked with FORMSPECWARNJCW. An error will cause one of the following actions:

- If any batch mode command except the FILE command causes an error, then FORMSPEC will skip that command and continue to the next command.
- If a FILE command caused the error or the error occurred because FORMSPEC did not recognize a batch mode command, then all batch mode commands, except the FORMS command, will be skipped until the next FILE or the EXIT command is encountered.

Batch mode output (errors, warnings, or messages) will normally be directed to $STDLIST. However, output can be redirected to another file by using the MPE :FILE command to redefine the formal file designator, FORMOUT. For example, since $STDLIST in a session is the terminal, you may want to redirect output to the line printer as follows:

```
:FILE FORMOUT;DEV=LP
```

When the output file is redirected to a disc file or the list device, the batch mode commands will also be echoed to the output file as they are executed. This will help to determine the cause of any errors that may occur.

BATCH MODE COMMANDS

The following pages show the syntax and description of the batch mode commands that FORMSPEC accepts in a command file or job stream. Table 7-1. gives a summary of the batch mode commands, a brief description of their functions, and the interactive counterpart for each command.

Note that each command must be entered on a separate line and a command may not carry over to the next line. Leading blanks may be used for indentation. You can also use blank lines between commands to make the command file easier to read. Comments may be inserted anywhere in a command file. They may be more than one line long, but they must be delimited by double arrows *comment*.

Table 7-1. FORMSPEC in Batch Mode Commands

Command	Function	Interactive Counterpart
Batch Mode Commands		
`COMPILE`	Compiles the current forms file.	Selection **X** on the FORMSPEC Main Menu.
`COPY`	Copies forms within the current forms file, or from another file to the current forms file.	Selection **C** on the FORMSPEC Main Menu.
`DELETE`	Deletes a form from the current forms file.	Selection **D** on the FORMSPEC Main Menu.
`EXIT`	Terminates FORMSPEC in batch mode.	Function key **f8** in any menu.
`FIELD`	Updates field attributes.	Field attributes specified on the FORMSPEC Field Menu.
`FILE`	Declares the current forms file.	Forms file name on the Forms File Menu.
`FKLABELS`	Creates or updates function key label strings.	FORMSPEC form-specific Function Key Menu.
`FORMS`	Prints summary information on the current forms file as well as for each or all the forms therein.	No interactive counterpart for this command.
`LIST`	Lists a form or all the forms on the standard list device.	Selection **L** on the FORMSPEC Main Menu.
`RELATE`	Establish family relation between two forms.	Selection **R** on the FORMSPEC Main Menu.
`RENUMBER`	Renumbers fields within a form.	No interactive counterpart for this command.

Table 7-1. FORMSPEC in Batch Mode Commands

Command	Function	Interactive Counterpart
SELECTLANG	Updates native language specifications.	FORMSPEC Terminal/Language Menu.
SELECTTERM	Updates terminal and device specifications.	FORMSPEC Terminal/Language Menu.
ARB Commands		
ARBTOSCREEN	Set data type conversion defaults from ARB to screen.	Select defaults on FORMSPEC Data Type Conversion Menu.
SCREENTOARB	Sets data type conversion rules from screen to ARB.	Set defaults on FORMSPEC Data Type Conversions Menu.
DELARB	Deletes an ARB.	Select DEL on the FORMSPEC ARB Menu.
GENARB	Generates an ARB from a form.	Select GEN on the FORMSPEC ARB Menu.
ADDARBFIELD	Adds a field to the ARB.	Select ADD on the FORMSPEC Restructure ARB Menu.
DELARBFIELD	Deletes a field from the ARB.	Select DEL on the FORMSPEC Restructure ARB Menu.
MODARBFIELD	Modifies the attributes an ARB.	Select MOD on the FORMSPEC ARB Menu.
MOVEARBFIELD	Moves a field or group of fields around on the ARB.	Select MOVE on the FORMSPEC Restructure ARB Menu.
RENAMEARBFIELD	Changes the name of an ARB field.	Select RENAME on the FORMSPEC Restructure ARB Menu.

COMPILE

Compiles the current forms file.

Syntax

```
COMPILE [INTO fastfile]
```

Parameters

fastfile	is the name of the fast forms file to which all code records are copied. Since a fast forms file contains only records needed by VPLUS/V intrinsics at run-time, it may be advantageous to compile the forms file into a fast forms file. If *fastfile* already exists and it is a fast forms file, it is replaced with this version. If *fastfile* already exists and it is not a fast forms file, an error occurs. The current forms file still compiles, but the code records are not copied to a fast forms file. If *fastfile* does not exist, it is created for you as a fast forms file.

Discussion

Normally the `COMPILE` command only compiles the forms that have been modified in the current forms file. However, when certain global attributes (such as save fields) are modified, all forms in the current forms file are compiled. The fast forms file, if specified, contains code records for all forms in the current forms file.

Example

```
COMPILE
```

compiles the current forms file.

```
COMPILE INTO INVFFORM.PUB
```

compiles the current forms file and creates a fast forms file named `INVFFORM` in the `PUB` group.

COPY

Copies a form within the current forms file, or copies a form from another forms file to the current forms file.

Syntax

```
[COPY form {TO newform
            IN file [TO newform]}]
```

Parameters

form	is the name of the form to be copied.
newform	is the name of the new form in the current forms file to which *form* will be copied. The *newform* cannot already exist in the forms file. If *newform* already exists, an error will occur and no copying will be done. If TO *newform* is omitted, the new form created in the current forms file will have the same name as the form it was copied from.
file	is the name of the forms file, if *form* is not in the current forms file.

Example

```
COPY PARTS TO PARTS2
```

copies FROM the form PARTS in the current forms file TO the new form PARTS2.

```
COPY PARTS IN INVFORMS TO PARTS2
```

copies FROM the form PARTS in the forms file INVFORMS TO the new form PARTS2 in the current forms file.

```
COPY PARTS IN INVFORMS
```

copies FROM the form PARTS in the forms file INVFORMS TO the new form PARTS in the current forms file.

DELETE

Removes a form from the current forms file.

Syntax

```
        DELETE form
```

Parameters

form	is the name of the form to be removed from the current forms file. If *form* does not exist, FORMSPEC will issue a warning and proceed to the next command.

Example

```
     DELETE ORDER2
```

removes the form `ORDER2` from the current forms file.

EXIT

Terminates FORMSPEC in batch mode.

Syntax

```
EXIT
```

Discussion

The `EXIT` command is optional. FORMSPEC will also terminate after the last command in the command file is executed.

FIELD

Updates field attributes.

Syntax

```
                         {,newfieldname}
                         {,enhancement }
FIELDform {fieldtag    } {,fieldtype   }
          {oldfieldname} {,datatype    }
                         {,initialvalue}
```

Parameters

form	is the name of an existing form within the current forms file.
fieldtag	tag assigned to the field in form screen design.
oldfieldname	name currently assigned to the field in the field menu.
newfieldname	replacement field name.
enhancement	replacement display enhancement code.
fieldtype	replacement field type.
datatype	replacement data type.
initialvalue	replacement initial value. If $EMPTY is entered in the first six positions of the parameter, initial value is cleared.

Discussion

The first two parameters of this command identify the field to be updated. The remaining parameters are the actual updates. The field menu update parameters (*newfieldname*...*initialvalue*) may be longer than the corresponding number of character positions available on the Field Menu in interactive FORMSPEC, in which case the input parameter is truncated (i.e., treated as command documentation).

Example

```
FIELD PARTS a_b_ c A _ B_C_CODE,NONE,R,,D
```

Updates field menu selections for field a_b_c of form PARTS. Specifically, the field name is set to A_B_C_ CODE, the field display enhancement code is set to NONE (for no enhancements), the field type value is set to R (for required), and the data type value is unchanged. The initial value for the field is set to the character D.

FILE

Specifies the current forms file to be used.

Syntax

```
        FILE file
```

Parameters

file	is the name of the forms file to be used by subsequent batch mode commands. The *file* is referred to as the current forms file.

Discussion

The FILE command must precede all other commands in a command file or job stream. This command opens the specified forms file to be operated on by the other batch mode commands. Only one forms file can be modified at a time (only one current forms file at a time). However, more than one forms file can be operated on in one command file or job stream as long as a FILE command precedes any operations on that file. Note that a FILE command specifying a new current forms file closes the previous current forms file and opens the new current forms file. Termination of FORMSPEC automatically closes the current forms file.

If the file specified does not already exist, it is created. A message indicating that the file was created is issued.

Example

```
     FILE INVFORMS.PUB
```

opens the forms file INVFORMS in the PUB group to be operated on by other batch mode commands.

FKLABELS

Updates function key labels.

Syntax

```
        FKLABELS form [\keylabel ... \keylabel]
```

Parameters

form	is the name of an existing form within the current forms file.
keylabel	literal which is to appear in the function key label. If $EMPTY is entered as the first six characters of *keylabel*, the existing key label is cleared.

Discussion

The FKLABELS command creates (if none exist) or updates function key labels for the specified form. The first 16 characters of the *keylabel* literal are used for the label. Only the first eight *keylabels* are used.

Example

```
      FKLABELS STAF_SALARY  Start      Over$EMPTY Exit Planner
```

Updates labels for function keys 2, 5 (which is cleared), and 8 in form STAFF_SALARY. Existing labels for function keys 1,3,4,6, and 7 are not updated.

FORMS

Prints a summary listing for the forms in the current forms file.

Syntax

```
[FORMS [form]
       [@ ]]
```

Parameters

form	The name of an existing form within the current forms file for which summary information is to be printed. If omitted, a summary of the current forms file is printed.
@	The commercial at sign (@) specifies that summary information for all forms, as well as the current forms file, is printed.

Discussion

The FORMS command prints the name of each form, the number of fields in the form, the number of bytes in the screen image, the number of children that the form has, and the name of the form's parent form, if any. If a form name (*form*) is included with the FORMS command, an extended summary is generated for the form. The extended summary includes all of the form information listed above. In addition, for each field in the form, the FORMSPEC assigned field number, data buffer offset (in bytes, one relative), and field length (in bytes) is printed. If the commercial at sign (@) is used for *form*, an extended summary is generated for each form in the forms file. See the examples below for samples of the three types of summary listings available.

Although all FORMSPEC output is normally directed to $STDLIST, you may want to redirect this output separately to the line printer. To do so, use the MPE :FILE command to redefine the formal file designator, FORMOUT, as follows:

```
:FILE FORMOUT;DEV=LP
```

The FORMS command prints a maximum of 60 lines per page. This maximum can be changed by setting the JCW.

```
FORMSPECLINESPERPAGE
```

See Appendix E for more information.

The FORMS command is a very useful bath mode command. When FORMS is used after the forms file has been updated, the listing can be used to quickly check that the resulting forms in the file are the correct ones.

This command also provides useful reference information when coding (or modifying) an application data buffer that has to match the layout of the fields in a form.

Example

```
FORMS
```

prints a summary of the current forms file, SYSADMIN in this example:

```
Forms file: SYSADMIN.ACCTG.FINANC                                 Page 1
File last modified: WED, MAY 14, 1986, 1:58 PM
File last compiled: WED, MAY 14, 1986, 1:58 PM

             NUMBER LARGEST                  NUMBER OF:
               OF    FIELD                   BYTES IN
FORM NAME    FIELDS  NUMBER    NEXT FORM     SCREEN  CHILDREN PARENT FORM
-----------  ------  ------  -----------    -------  -------- -----------
ACCOUNT_MGRS   5        5    $HEAD            114        0
SYSTEMZ-USERS  5        5    @HEAD             98        0
```

Example

```
FORMS SYSTEM_USERS
```

Prints a summary of the form specified, SYSTEM_USERS in this example:

```
Forms File: SYSADMIN.ACCTG.FINANC                                 Page 1
File last modified: WED, MAY 14, 1986, 1:58 PM
File last compiled: WED, mAY 14, 1986, 1:58 PM

             NUMBER LARGEST                   NUMBER OF:
               OF    FIELD                   BYTES IN
FORM NAME    FIELDS  NUMBER    NEXT FORM     SCREEN  CHILDREN PARENT FORM
-----------  ------  ------  -----------    -------  -------- -----------
SYSTEM_USERS    5       5    $HEAD             98        0

Forms File: SYSADMIN.ACCTG.FINANC                                 Page 2

             SCREEN  BUFFER
             LENGTH  LENGTH                        FIELD   LENGTH   BUFFER
FORM NAME    (bytes) (bytes)   FIELD NAME          NUMBER  (bytes)  OFFSET
-----------  ------  ------  -----------------    ------- ------- -------
SYSTEM_USERS   168     33    **** Compiled for: HP264X
                             USER_NAME                1       8        0
                             ACCOUNT_NAME             2       8        8
                             HOME_GROUP               3       8        16
                             PORT_NUMBER              4       4        24
                             PHONE_EXTENSION          5       5        28

****End of compiled forms listing. ****
```

Example

```
FORMS @
```

Prints a summary of the current forms file, (SYSADMIN) as well as the summary information for each form in the forms file, two in this example:

```
Forms File: SYSADMIN.ACCTG.FINANC                                 Page 1
File last modified: WED, MAY 14, 1986, 1:58 PM
File last compiled: WED, mAY 14, 1986, 1:58 PM

             NUMBER LARGEST                  NUMBER OF:
               OF    FIELD                   BYTES IN
FORM NAME    FIELDS  NUMBER    NEXT FORM     SCREEN  CHILDREN PARENT FORM
-----------  ------  ------  -----------    -------  -------- -----------
ACCOUNT_MGRS    5       5    $HEAD            114        0
SYSTEM_USERS    5       5    $HEAD             98        0

Forms File: SYSADMIN.ACCTG.FINANC                                 Page 2

             SCREEN  BUFFER
COMPILED     LENGTH  LENGTH                      FIELD   LENGTH   BUFFER
FORM NAME    (bytes) (bytes)   FIELD NAME        NUMBER  (bytes)  OFFSET
-----------  ------  ------  ------------------  ------- ------- -------
ACCOUNT_MGRS   155     68    **** Compiled for: HP264X
                             ACCOUNT_NAME           1       8       0
                             MGR_LAST_NAME          2       20      8
                             MGR_FIRST_NAME         3       15      28
                             DEPARTMENT             4       20      43
                             PHONE_EXTENSION        5       5       63
SYSTEM_USERS   168     33    **** Compiled for: HP264X
                             USER_NAME              1       8       0
                             ACCOUNT_NAME           2       8       8
                             HOME_GROUP             3       8       16
                             PORT_NUMBER            4       4       24
                             PHONE_EXTENSION        5       5       28

****End of compiled forms listing. ****

 Number of errors = 0   Number of warnings = 0
```

LIST

Lists a specified form or all forms on the list device.

Syntax

```
LIST [form]
```

Parameters

form	is the name of the form in the current forms file to be listed. When *form* is not specified, all forms in the current forms file are listed.

Discussion

The LIST command can be used to provide documentation for a forms file. The forms are usually listed on the line printer. The LIST command can also be used to list forms to a disc file to be examined. To do so, use the MPE :FILE command to redefine the formal file designator FORMLIST as shown below:

```
:FILE FORMLIST=MYFILE;DEV=DISC;SAVE;NOCCTL
```

will list forms to the disc file, MYFILE. The SAVE keyword specifies that MYFILE be saved as a permanent file. The NOCCTL keyword indicates that no carriage control characters are written to the file. (See the *MPE Commands Reference Manual*(30000-90009), for more information on the MPE:FILE command.)

The LIST command prints a maximum of 60 lines per page. This maximum may be changed by setting the JCW.

```
FORMSPECLINESPERPAGE
```

See Appendix E for more information.

Example

```
LIST
```

lists all forms in the current forms file on the list device.

```
LIST PARTS
```

lists the form PARTS in the current forms file on the list device.

RELATE

Creates a family relationship between two existing forms.

Syntax

```
    RELATE childform TO parentform
```

Parameters

childform	is the name of the form which is to become the child form. It must already exist, and not be related to any other form.
parentform	is the name of the form which is to become the parent form. It must already exist, and have the identical form design as the childform specified with the *childform* parameter.

Example

```
RELATE FORM1 TO FORM2
```

Establishes a family relationship between FORM1 and FORM2 where FORM1 is the child form, and the FORM2 is the parent form.

RENUMBER

Renumber fields to screen order.

Syntax

```
RENUMBER form
```

Parameters

form	is the name of the form containing the fields to be renumbered.

Discussion

The fields in the form specified by *form* are renumbered in accordance with screen order, left to right, top to bottom. If your application depends on field numbers, be sure to update your application to reflect the changed field numbers. If the fields are already numbered in screen order, a message is returned and the fields are not renumbered.

For family forms, the parent form must be specified; all family members are then automatically renumbered. Again, if your application depends on field numbers, be sure to update your application, in this case, for all family members. If you specify the name of a child form, the fields are not renumbered. Instead, a message is returned stating the form was not renumbered because it is a child form and indicating the name of the parent form.

Example

```
RENUMBER FORM1
```

Renumbers the fields in `FORM 1` in screen order.

SELECTLANG

Updates native language specification.

Syntax

```
        SELECTLANG      [formsfilelangID] [,FORMSPEClangID]
```

Parameters

formsfilelangID Set forms file native language specification.

FORMSPEClangID Set FORMSPEC native language specification.

Discussion

Only the first three characters of each parameter are used. Refer to "Setting Language ID Number" in Section 8 for more information. At least one parameter must be specified.

If no parameters are provided, existing native language specifications are retained,

Example

```
     SELECTLANG 8
```

Native language ID 8, German, is specified for the forms file.

```
     SELECTLANG ,O
```

Native language ID 0, Native-3000, is specified for FORMSPEC. The native language specified for the forms file is retained.

SELECTTERM

Updates forms file terminal or device specifications.

Syntax

```
          SELECTTERM      [262xand239x] [,color] [,264x] [,307x]
```

Parameters

262xand239x	Configure forms file for the HP 262X and HP'239X terminal families; also includes the HP 150.
color	Configure forms file for color terminals.
264x	Configure forms file for the HP 264X terminal family. Specifying yes (the value Y or y) enables function key labeling.
307x	Configure forms file for the data capture device family. Specifying eight (the value 8) enables support of the extended features of the HP 3081A.

Discussion

The parameters may be any valid non-blank character to select the terminal family. Only the first character of each parameter is processed (the rest is treated as command documentation).

At least one parameter must be specified. Providing a USASCII blank as the specification for a given device/terminal family does not change the existing specification. Specifying $EMPTY clears an existing specification.

Example

```
        SELECTTERM x,,y
```

Forms file is configured for HP 262X/239X terminals, the HP 150, and HP 264X terminals, with HP 264X function key labeling enabled. Existing color specification, if any, is retained.

```
        SELECTTERM ,x,$EMPTY,8
```

Forms file is configured for color terminals and HP 307x terminals, with HP 3081A features enabled. HP264X specification is cleared and existing HP262X/239X/150 specification, if any, is retained.

ARB BATCH MODE COMMANDS

The following batch mode commands support the ARB feature. They are based on the ARB Menu options provided in interactive FORMSPEC. The commands are summarized below, and each one is described in detail in the rest of this section.

NOTE In the ARB commands, *`destination`* is:

[`BEFORE fieldname`] or [`AFTER fieldname`]

The data type conversion commands are:

`ARBTOSCREEN and ARBTOSCREEN`

The ARB commands are:

`DELARB and GENARB`

The ARB field commands are:

`ADDARBFIELD, DELARBFIELD, MODARBFIELD, MOVEARBFIELD, and RENAMEARBFIELD`

Their functions are summarized in Table 7-1.

ADDARBFIELD

Add a field to the ARB.

Syntax

```
ADDARBFIELD arbname fieldname destination
```

Parameters

`arbname`	Name of the ARB to which the field will be added.
`fieldname`	The name of the new field.
`destination`	Position at which the field is to be inserted. May be left blank.

Discussion

If the *`arbname`* doesn't exist, an ARB with this name will be created for the associated form. The ARB cannot contain the *`fieldname`* already. If *`fieldname`* is not a field on the form, it becomes a filler field, with no datatype and with a length of 1 (this length may be changed using the `MODARBFIELD` command). If *`fieldname`* is on the form, its ARB *`length`* and *`data type`* are created from its screen attributes and the Data Conversion table (set up using the Data Conversion Menu). If *`destination`* is specified, the field will be inserted at that position, otherwise it will be appended to the end of the ARB.

ARBTOSCREEN

Create/update the datatype conversion rules for converting ARB field contents to screen field contents.

Syntax

```
     ARBTOSCREEN [chartype, inttype, realtype, packtype, zonetype]
```

Parameters

chartype	The conversion type for ARB fields with type CHAR.
inttype	The conversion type for ARB fields with type INT or DINT.
realtype	The conversion type for ARB fields with type REAL or LONG.
packtype	The conversion type for ARB fields with type PACK, PACK0, PACK1,... PACK9, SPACK, SPACK0, SPACK1,...SPACK9.
ZONETYPE	The conversion type for ARB fields with type ZONE, ZONE0, ZONE1... ZONE 9.

Discussion

This command creates/updates the datatype conversion rules for converting ARB data types to screen data type. If a data type update is omitted, then the current default for that type is retained. If no update parameters are specified, all ARB-to-screen conversion types are set to CHAR, the default. Leading commas must be supplied.

Valid data type updates are: CHAR, YMD, MDY, DMY, DIG, NUM(*n*) NUMA, IMP*n* IMPA

EXAMPLE

```
 The command:

     ARBTOSCREEN, DIG, NUM
```

Sets the conversion type for INT and DINT ARB fields to DIG and for REAL and LONG ARB fields to NUM.

DELARB

Deletes the entire specified ARB.

Syntax

```
     DELARB arbname
```

Parameters

arbname	`Name of the existing ARB.`

Discussion

All the fields are deleted. You can generate another ARB for the specified form if required. If extensive changes are made to a form, it is easier to delete the existing ARB and create a new one than to revise the old ARB.

DELARBFIELD

Deletes a specified field from the ARB.

Syntax

```
DELARBFIELD arbname fieldname
```

Parameters

arbname	Name of the existing ARB.
fieldname	Name of the specified field on the ARB.

Discussion

The *arbname* must exist in the forms file and the *fieldname* must exist on the ARB.

GENARB

Generates an ARB from the form specified by the formname.

Syntax

```
GENARB arbname
```

Parameters

arbname	Same as the name of the associated form

Discussion

The ARB will contain a field for every field on the form. The data type and length of each ARB field are determined from the screen attributes and the Data Conversion table (set up using the Data Conversion Menu).

MODARBFIELD

Modify the attributes of a field on an ARB.

Syntax

```
MODARBFIELD arbname fieldname {[length, type]}
```

Parameters

arbname	Name of the ARB that contains the field to be modified
fieldname	The name of the field to be modified.
[length, type]	The new ARB field length and/or data type.

Discussion

The *arbname* must exist in the forms file and the *fieldname* must exist on the ARB. You can change both the parameters *length* and *type*, but you must specify at least one. They must conform to the edit rules (see Section 3).

EXAMPLE

```
The command:

    MODARBFIELD EMPLOYEE SSNUM, CHAR

changes the ARB data type to CHAR.
```

MOVEARBFIELD

Move a field or group of fields to a specified position on an ARB.

Syntax

```
     MOVEARBFIELD arbname range [destination]
```

Parameters

arbname	The name of the ARB.
range	The field or fields to be moved, expressed as fieldname or fieldname/fieldname.
[destination]	The position in the ARB to which the field(s) will be moved. Can be left blank.

Discussion

The *arbname* must exist. If the *destination* is not specified, the field(s) will be appended at the end of the ARB.

EXAMPLE

Suppose the fields in the ARB EMPLOYEE are in this order:

```
NAME
ADDRESS
PHONE
SSNUM
SUPERVISOR
```

The command:

```
     MOVEARBFIELD EMPLOYEE ADDRESS/PHONE AFTER SSNUM
```

or the command:

```
     MOVEARBFIELD EMPLOYEE SSNUM BEFORE ADDRESS
```

will rearrange the fields in this order:

```
NAME
SSNUM
ADDRESS
PHONE
SUPERVISOR
```

RENAMEARBFIELD

Change the name of *oldfieldname* to *newfieldname* in the ARB *arbname*.

Syntax

```
RENAMEARBFIELD arbname oldfieldname newfieldname
```

Parameters

arbname	Name of the ARB that contains the field.
oldfieldname	The current name of the field.
newfieldname	The new game you want to give the field.

Discussion

This command does not change the ARB data type or length. The *arbname* must exist in the forms file, and the *oldfieldname* must exist on the ARB. The ARB cannot already contain a field with the *newfieldname*. If *newfieldname* is not a field on the form, the renamed field becomes a filler field.

SCREENTOARB

Create/update the datatype conversion rules for generating an ARB from a form.

Syntax

```
SCREENTOARB [chartype, datetype, digtype, numtype, imptype]
```

Parameters

chartype	The conversion type for screen fields with type CHAR.
datetype	The conversion type for screen fields with a date type.
digtype	The conversion type for screen fields with type DIG.
numtype	The conversion type for screen fields with type NUM, NUM0, NUM1… NUM9.
imptype	The conversion type for screen fields with type IMP, IMP0, IMP1,… IMP9.

Discussion

This command creates/updates the datatype conversion rules for generating an ARB from a form. If an update parameter is omitted, then the current conversion type for that parameter is retained. If no update parameters are specified, all screen-to-ARB conversion types are set to CHAR, the default. Leading commas must be supplied.

Valid type updates are: CHAR, YYMMDD, INT, DINT, REAL, LONG, PACK*n*, SPACK*n*, ZONE*n*, PACKA, SPACKA, ZONEA.

EXAMPLE

The command:

```
SCREENTOARB, INT, REAL, REAL
```

sets the conversion type for DIG screen fields to INT, and for NUM and IMP screen fields to REAL.

SAMPLES OF FORMSPEC IN BATCH MODE

The following are samples of how FORMSPEC in batch mode is used, first with a command file and then with the commands directly in the job stream.

Sample Command File

FORMSPEC in batch mode can be executed with the following commands:

```
:FILE FORMOUT;DEV=LP
:RUN FORMSPEC.PUB.SYS;INFO="FORMCMDS.PUB.MFG"
```

The MPE :FILE command directs batch mode output (commands and error messages) to the list device. The MPE :RUN command with the INFO=*filename* parameter executes FORMSPEC in batch mode, taking batch mode commands from the command file FORMCMDS.PUB.MFG. Remember, you must first create the command file using a text editor.

Sample Job Stream

To stream a job using FORMSPEC in batch mode, you must first create the job file using a text editor. Then, stream the job using the MPE :STREAM command.

The following is a listing of a sample job stream which compiles a forms file and documents its contents:

```
!JOB FORMUPDT, MIKE.MFG,FORMS
!FILE FORMOUT; DEV=LP
!RUN FORMSPEC. PUB.SYS
FILE FORMSF1
COMPILE
FORMS
EXIT
!EOJ
```

Example

```
<< This command file updates the forms files, FORMSF1 and FORMSD2.          >>
<< FORMSF1 is opened first. Two forms are copied from the forms file        >>
<< MASTERFS.PUB. The FORMSF1 is then compiled into the fast forms file      >>
<< FASTFRM1. A listing is made of all the forms in FORMSF1.                 >>
<< FORMSF1 is then closed and FORMSF2 opened as the current forms file.     >>
<< Two forms are copied from the forms file MASTERFS.PUB to new forms       >>
<< in the current forms file. FORMSF2 is then compiled (note that NO        >>
<< fast forms file is produced). The names of the fomrs in FORMSF2 are      >>
<< printed after all modifications to the forms file are complete.          >>

FILE FORMSF1                          << Opens the forms file to be updated  >>

DELETE FORMA                          << Deletes forms to current forms file >>
DELETE FORMB                          << in case they already exist.         >>

COPY FORMA INTO MASTERFS.PUB          << Copies forms to current forms file  >>
COPY FORMB INTO MASTERFS.PUB          << using same form names.              >>

COMPILE INTO FASTFRM1                 << Compiles current forms file to      >>
                                      << fast forms file.                    >>

LIST                                  << Lists the forms in FORMSF1.         >>

FILE FORMSF2                          << Closes FORMSF1 and opens FORMSF2    >>
                                      << as new current forms file.          >>

DELETE FORM2A                         << Deletes forms in current forms file >>
DELETE FORM2B                         << in case they already exist.         >>

COPY FORMA IN MASTERFS.PUB TO FORM2A  <<Copies forms to new current forms    >>
COPY FORMB IN MASTERFS.PUB TO FORM2B  << file using new form names.          >>

COMPILE                               << Compiles current forms file.        >>

FORMS                                 << Prints the names of the forms in    >>
                                      << the current forms file.             >>

EXIT                                  << Closes FORMSF2 and terminates       >>
                                      << FORMSPEC in batch mode.             >>
```

8 NATIVE LANGUAGE SUPPORT

VPLUS Native Language Support (NLS) enables an applications designer to create interactive user applications which reflect both the user's native language and the local custom for numeric and date information in the supported languages. (See *Native Language Support Reference Manual* for an explanation of supported languages.) NLS provides these specific features in VPLUS:

- Native decimal and thousands indicators.
- Native language month names for dates.
- Alphabetic upshifting of native characters.
- Native characters in single value comparisons, table checks, and range checks.
- Native collating sequence in range checks and single value comparisons.

VPLUS does not support the application design process in native languages. Form names, field identifiers and field tags support only USASCII characters.

REFSPEC and REFORMAT do not use NLS features. These programs interact with users in NATIVE-3000 only.

LANGUAGE ATTRIBUTE

VPLUS contains an NLS native language attribute option which allows the applications programmer to design a forms file which reflects the native language characteristics of the application. Each forms file has a global native language ID number. The application may be unlocalized, language-dependent, or international. Refer to the *Native Language Support Reference Manual* for a list of the available native language ID numbers and for examples of these applications.

Unlocalized (NATIVE-3000)

If a native language ID number is not assigned to a forms file, it will default to `0` (NATIVE-3000). NLS features do not impact an unlocalized forms file.

Language-Dependent

Language-dependent forms files are used when the associated application only operates in a single native language context. The native language ID number for the specific native language desired is assigned when the forms file is designed.

International

Multinational corporations may need to maintain a business language for commands, titles, and menus in addition to accommodating the native language of the user for the actual data retrieved or displayed. For this application, `-1` is selected as the native language ID number for the forms file. The VPLUS intrinsic VSETLANG must be called at run-time to assign the appropriate native language ID.

NLS CAPABILITIES

The components of a form which can be language-dependent are the initial values of fields and the field edit rules. The text is fixed in a single native language by the forms designer. The native language ID number determines the context for data editing, conversion, and formatting. There are two native language IDs assigned for each forms file. The FORMSPEC language controls the context when the forms file is designed. The forms file language controls the context when the forms file is executed.

Setting the Native Language ID Numbers

The forms designer sets both native language ID number values for the forms file via the FORMSPEC Terminal/Language Selection Menu. NATIVE-3000 is currently the only selection available for FORMSPEC language. This means that initial values and processing specifications must be defined with the month names and numeric conventions of NATIVE-3000.

The native language ID for the forms file language defaults to 0 (NATIVE-3000) if no native language ID number is specified on the Terminal/Language Selection Menu. On this menu, the forms designer can assign or change the native language ID for the forms file language at any time. The value specified must be a positive number or a zero for a single native language application. If the value specified is acceptable, but the native language is not configured on the system used for forms design, FORMSPEC issues a warning message but does not reject the native language ID number. Instead, the designer is prompted to confirm the value or change it.

For applications that are used with multiple native languages, the forms designer specifies the international language ID number `-1`. The international language ID number allows the intrinsic VSETLANG to be called at run-time to select the actual native language ID number for the forms file. If an application uses an international forms file without calling VSETLANG, it is executed in the default, NATIVE-3000. If VSETLANG is called for an unlocalized or language-dependent forms file, an error code will be returned.

The designer has three options in designing an application to work effectively with multiple languages:

- Develop several language-dependent forms files.
- Create one international forms file.
- Produce a combination of language-dependent files and an international forms file.

If the text needs to be in the native language, unique versions of a forms file are required for each native language supported.

VGETLANG may be used to determine whether a language-dependent forms file or an international forms file is being executed. If VGETLANG indicates an international forms file, VSETLANG must be called to select the actual native language. Refer to the VGETLANG and VSETLANG intrinsics in Section 6.

FIELD EDITS

NATIVE-3000 must be used to specify date and numeric fields within FORMSPEC. When the forms file is executed, VPLUS will convert the value to be consistent with the native language selected. Single value comparisons (GT,GE,LT,LE,EQ,NE), as well as table and range checks (IN, NIN), specified within FORMSPEC may contain any character in the 8 bit extended character set consistent with the selected native language ID number. When the form is executed at run-time, the collating table for the native language specified is used to check whether the field is within a range.

Date Handling

VPLUS supports several date formats and three date orders: MDY, DMY,YMD. Any format is acceptable as input when the form is executed, provided that the field length can accommodate the format. The forms designer specifies the order of each date-type field. With NLS, the alphabetic month names are edited and converted to numeric destinations using the month names corresponding to the native language of the forms file. The format and the date order are not related to the native language of the forms file.

Numeric Data

Decimal and thousands indicators are native language-dependent in the NUM[n] and IMPn fields. When data is moved between fields and automatic formatting occurs for data entered in any field, recognition, removal or insertion of these decimal and thousands indicators is native language-dependent. The optional decimal symbol in constants is also native language-dependent.

NOTE VPLUS edit processing specifications and terminal edit processing statements are separate and are not checked for compatibility. There will be no check to see that the forms designer has specified a terminal local edit (DEC_TYPE_EUR, DEC_TYPE_US) in the configuration phase which is consistent with the native language-dependent decimal indicator for the native language specified in the forms file.

Native Language Characters

If a native language ID number has been specified in the forms file, the `UPSHIFT` formatting statement uses native language upshift tables.

Range checks and the single value comparisons GT, GE, LT and LE involve collating sequences. When the form is executed, the native language collating sequence table designated by the native language ID number is used to check whether the field passes the edit.

NLS features in VPLUS do not include support for pattern matching with native characters. MATCH uses USASCII specifications.

ENTRY AND LANGUAGE lD NUMBER

The native language ID assigned for the forms file language determines the native language used by ENTRY unless the file is international (`-1`). ENTRY uses the intrinsic `VGETLANG` to identify the native language ID assigned by the forms designer for the forms file language.

If the forms file is international, ENTRY calls the NLS intrinsic `NLGETLANG` (mode 1). If it returns a value of UNKNOWN, the user is prompted for a native language ID number. Once a valid native language ID number is determined, ENTRY calls the `VSETLANG` intrinsic to specify the corresponding native language.

The batch file does not have a native language indicator. Users with different native languages may collect data in the same batch file if the associated forms file is international.

A SAMPLE PROGRAMS

The VPLUS intrinsics can be called from the programming languages listed in Table A-1. The reference manuals to consult for each language are also listed.

Table A-1. Programming Languages and References

Language	Reference Manual
BASIC HP BUSINESS BASIC	*BASIC/300O Interpreter Reference Manual* *BASIC/3000 Compiler Reference Manual*, *HP BUSINESS BASIC Reference Manual*, *HP BUSINESS BASIC Programming Guide*
COBOL	*COBOL/3000 Reference Manual*, *COBOLII/3000 Reference Manual*, *COBOLII/3000 Conversion Guide*, *Using COBOL*
FORTRAN	*FORTRAN/3000 Reference Manual*, *HP FORTRAN 77 Reference Manual*, *FORTRAN 77 Supplement*
Pascal	*Pascal/3000 Reference Manual*
SPL	*System Programming Language Reference Manual*
TRANSACT	*Transact/3OOO Reference Manual Getting Started with Transact Manual*

In addition, this appendix contains sample programs, with the VPLUS intrinsic calls highlighted, for each of the following programming languages:

Language	Page
SPL	406
COBOL	430
FORTRAN 77	442
BASIC	461
TRANSACT	469
PASCAL	478

SPL

```
$PAGE "HP32209B.04.17 VPLUS/V S40S209B, ENTRY"
$COPYRIGHT "                                                          " , &
$ "                                                               " , &
$ "       (c)     COPYRIGHT HEWLETT-PACKARD.       1986           " , &
                                                                  " , &
$ "This program may be used with one computer system at a time    " , &
$ "and shall not otherwise be recorded, transmitted or stored     " , &
$ "in a retrieval system. Copying or other reproduction of this   " , &
$ "program except for archival purposes is prohibited without     " , &
$ "the prior written consent of the Hewlett-Packard Company.      " , &
<<                                                                >>
$CONTROL USLINIT, LIST, MAP, CODE
<<****************************************************************>>
<<                                                                >>
<<                 ENTRY--VPLUS/V Data Entry Program              >>
<<                                                                >>
<<                             9/1/79                             >>
<<                                                                >>
<<****************************************************************>>
<<
   This program controls source data entry for any forms file.
   It opens a forms file, based on user input; it opens a batch
   file, also named by the user. If all is ok, it displays the
   head form, accepts input, edits the data, and if no errors,
   writes it to the batch file. The program continues to do this
   until $END is reached, or until the EXIT function key has been
   pressed.

   This program also controls browsing through the data collected,
   and supports modification of that data.

   The function keys have defined meanings as follows:


            f1         f2         f3         f4
           HEAD      DELETE     PRINT      REFRESH

            f5         f6         f7         f8
           PREV       NEXT      BROWSE/     EXIT
                                COLLECT

>>
```

```
$PAGE "           "ENTRY DECLARATIONS"
<<*****************************************************************>>
<<                                                                 >>
<<                  ENTRY Global Declarations                      >>
<<                                                                 >>
<<*****************************************************************>>
BEGIN
DEFINE
    VERSION              = " B.04.17" #
   ,ID'MSG=("HP32209",VERSION," ENTRY (C) HEWLETT-PACKARD CO. 1986")#
   ;
DEFINE
    COM'STATUS          = COMAREA (0) #
   ,COM'LANGUAGE        = COMAREA (1) #
   ,COM'COMAREALEN      = COMAREA (2) #
   ,COM'MODE            = COMAREA (4) #
   ,COM'LASTKEY         = COMAREA (5) #
   ,COM'NUMERRS         = COMAREA (6) #
   ,COM'LABEL'OPTION    = COMAREA (9) #
   ,COM'CFNAME          = COMAREA'B (10*2) #
   ,COM'NFNAME          = COMAREA'B (18*2) #
   ,COM'REPEATOPT       = COMAREA (26) #
   ,COM'NFOPT           = COMAREA (27) #
   ,COM'DBUFLEN         = COMAREA (29) #
   ,COM'DELETEFLAG      = COMAREA (32) #
   ,COM'SHOWCONTROL     = COMAREA (33) #
   ,COM'NUMRECS         = COMAREA'D (21) #
   ,COM'RECNUM          = COMAREA'D (22) #
   ,COM'TERMFILENUM     = COMAREA (48) #
   ,COM'TERMOPTIONS     = COMAREA (55) #
   ,com'term'type       = comarea (58) #
   ,com'keyboard'type   = comarea (74) #
   ,com'form'stor'size  = comarea (38) #
   ;
DEFINE
    CHECK'ERROR = IF COM'STATUS <>' 0 THEN
                     ERROR #
   ,CHECK'EDIT'ERROR = IF COM'STATUS <> 0 OR COM'NUMERRS <> 0 THEN
                          ERROR #
   ;
EQUATE    << MISCELLANEOUS VALUES >>
     COMAREALEN     =   85
    ,SPL'LANG      =   3
    ,COLLECT'MODE  =   0
    ,BROWSE'MODE   =   1
    ,MAXWINDOWLEN  =   150
    ,NAMELEN       =   15
    ,NORM          =   0
    ,NOREPEAT      =   0
    ,REPEAT        =   1
    ,REPEATAPP     =   2
    ,ESC           =   27
    ,FORWARDS      =   1

,BACKWARDS    = -1
     ;
EQUATE    << FUNCTION KEY ASSIGNMENTS >>
      ENTERKEY     =  0
     ,HEADKEY      =  1
     ,DELETEKEY    =  2
     ,PRINTKEY     =  3
     ,REFRESHKEY   =  4
     ,PREVKEY      =  5
     ,NEXTKEY      =  6
     ,BROWSEKEY    =  7
     ,EXITKEY      =  8
     ;
EQUATE    << ENTRY ERROR EQUATES >>
      PREV'NOT'ALLOWED       =  1
     ,NO'PREV'RECS           =  2
     ,NOT'REPEATING          =  3
     ,DELETE'NOT'DEFINED     =  4
     ,NO'BATCH'RECS          =  5
     ,NO'BATCH               =  6
     ,NO'NEXT'RECS           =  7
     ;
EQUATE << NATIVE LANGUAGE SUPPORT EQUATES >>
      INTERNATIONAL           = -1
     ;
INTEGER ARRAY
      COMAREA (O:COMAREALEN-1) := COMAREALEN (0)
     ;
BYTE ARRAY
      COMAREA'B (*) = COMAREA
     ;
DOUBLE ARRAY
      COMAREA'D (*) = COMAREA
     ;
LOGICAL
      ERRORS := FALSE
     ,BATCH
     ;
ARRAY
      MESSAGE'WBUF (0:MAXWINDOWLEN/2)
     ;
BYTE ARRAY
      MESSAGE'BUF (*) = MESSAGE'WBUF
     ;
INTEGER
      PARMVAL := 20
     ,UNDERLINE := 1
     ,MESSAGE'BUF'LEN := MAXWINDOWLEN
     ,MSGLEN
     ,PAGE'EJECT := %61
     ;
DOUBLE
      LAST'REC'NUM
     ;
```

```
$PAGE "           VPLUS/V INTRINSIC DECLARATIONS"
<<*****************************************************************>>
<<                                                                 >>
<<                     VPLUS/V INTRINSICS                          >>
<<                                                                 >>
<<*****************************************************************>>
INTRINSIC
      VCLOSEBATCH
     ,VCLOSEFORMF
     ,VCLOSETERM
     ,VERRMSG
     ,VFIELDEDITS
     ,VFINISHFORM
     ,VGETNEXTFORM
     ,VINITFORM
     ,VOPENBATCH
     ,VOPENFORMF
     ,VOPENTERM
     ,VPOSTBATCH
     ,VPRINTFORM
     ,VPUTWINDOW
     ,VREADBATCH
     ,VREADFIELDS
     ,VSHOWFORM
     ,VWRITEBATCH
     ,VGETKEYLABELS
     ,VSETKEYLABELS
     ,VSETKEYLABEL
     ,VSETLANG
     ,VGETLANG
     ;

<<*****************************************************************>>

<<                                                                 >>
<<                         DO'COLLECT'LABELS                       >>
<<                                                                 >>
<<*****************************************************************>>
PROCEDURE DO'COLLECT'LABELS;

  BEGIN

     BYTE ARRAY LABELS(0:127);

     INTEGER NUMBER'OF'LABELS,GLOB'FORM;

     MOVE LABELS := (
         << FUNCTION KEY 1 >>       "  HEAD   FORM    "
         << FUNCTION KEY 2 >>      ,"                 "
         << FUNCTION KEY 3 >>      ," PRINT           "
         << FUNCTION KEY 4 >>      ,"REFRESH          "
         << FUNCTION KEY 5 >>      ,"                 "
         << FUNCTION KEY 6 >>      ,"   NEXT   FORM   "
         << FUNCTION KEY 7 >>      ," BROWSE          "
<< FUNCTION KEY 8 >>        ," EXIT
                          );

      GLOB’FORM := 0; << GLOBAL LABELS >>

      NUMBER’OF’LABELS := 8;
      $$VSETKEYLABELS(COMAREA,GLOB’FORM,NUMBER’OF’LABELS, LABELS);

   END;

<<**************************************************************>>
<<                                                              >>
<<                      DO’BROWSE’LABELS                        >>
<<                                                              >>
<<**************************************************************>>
PROCEDURE DO’BROWSE’LABELS;

   BEGIN

      BYTE ARRAY LABELS(0:127);

      INTEGER NUMBER’OF’LABELS,GLOB’FORM;

      MOVE LABELS := (
           << FUNCTION KEY 1 >>                " FIRST      REC      "
           << FUNCTION KEY 2 >>               ," DELETE     REC      "
           << FUNCTION KEY 3 >>               ," PRINT               "
           << FUNCTION KEY 4 >>               ,"REFRESH              "
           << FUNCTION KEY 5 >>               ,"  PREV      REC      "
           << FUNCTION KEY 6 >>               ,"  NEXT      REC      "
           << FUNCTION KEY 7 >>               ," COLLECT             "
           << FUNCTION KEY 8 >>               ,"  EXIT               "
                          );

      GLOB’FORM := 0; << GLOBAL LABELS >>

      NUMBER’OF’LABELS := 8;

      VSETKEYLABELS(COMAREA,GLOB’FORM,NUMBER’OF’LABELS, LABELS);

   END;
```

```
$PAGE "           FORMAT'STATUS'LINE"
<<****************************************************************>>
<<                                                                >>
<<                        FORMAT'STATUS'LINE                      >>
<<                                                                >>
<<****************************************************************>>
PROCEDURE FORMAT'STATUS'LINE;
   BEGIN

   INTEGER CNT;

   INTRINSIC ASCII, DASCII;

   if com'term'type = 15 or << HP3075 >>
      com'term'type = 16 then << hp3076 >>
     move message'buf := (" ENTRY ", version, " "), 2
   else
      MOVE MESSAGE'BUF := (" ENTRY ", VERSION, ESC, "&a31C"), 2;
   MSGLEN := TOS - @MESSAGE'BUF;

   MOVE MESSAGE'BUF(MSGLEN) := "Batch Record #", 2;
   MSGLEN := TOS - @MESSAGE'BUF;
   MSGLEN := MSGLEN + DASCII (COM'RECNUM+1D, 10, MESSAGE'BUF (MSGLEN));

   if com'term'type = 15 or << hp3075 >>
      com'term'type = 16 then << hp3O76 >>
     move message'buf(msglen) := (" Mode: "), 2
   else
      MOVE MESSAGE'BUF (MSGLEN) := (ESC, "&a65CMode: "), 2;

   MSGLEN := TOS - @MESSAGE'BUF;
   IF COM'MODE = COLLECT'MODE THEN
        MOVE MESSAGE'BUF (MSGLEN) := "Collect", 2
   ELSE
      if com'term'type = 15 or << hp3075>>
         com'term'type = 16 then << hp3076 >>
        move message'buf(msglen):= ("Browse"), 2
      else
        MOVE MESSAGE'BUF (MSGLEN) : = (ESC, "&dKBrowse") , 2;

   MSGLEN := TOS - @MESSAGE'BUF;

   VPUTWINDOW (COMAREA, MESSAGE'BUF, MSGLEN);
   END;   << FORMAT'STATUS'LINE >>
```

```
$PAGE "           ENTRY’ERROR"
<<****************************************************************>>
<<                                                                >>
<<                        ENTRY’ERROR                             >>
<<                                                                >>
<<****************************************************************>>
PROCEDURE ENTRY’ERROR (ENTRY’ERROR’NUM);
VALUE ENTRY’ERROR’NUM;
INTEGER ENTRY’ERROR’NUM;
    BEGIN

    IF ERRORS THEN
       RETURN;

    ERRORS := TRUE;

    CASE ENTRY’ERROR’NUM OF
       BEGIN

       << 0 IS NOT DEFINED >>
          ;

       << PREV’NOT’DEFINED:      >>
          MOVE MESSAGE’BUF :=
              " The PREV key is only defined for browse mode.", 2;

       << NO’PREV’RECS:          >>
          MOVE MESSAGE’BUF :=
              " There are no previous batch records.", 2;

       << NOT’REPEATING:         >>
          MOVE MESSAGE’BUF :=
              " The NEXT key is not defined for a non-repeating form.", 2;

       << DELETE’NOT’DEFINED:    >>
          MOVE MESSAGE’BUF :=
              " The DELETE key is only defined for browse mode.", 2;

       << NO’BATCH’RECS:         >>
          MOVE MESSAGE’BUF :=
              " There are no batch records to browse.", 2;

       << NO’BATCH:              >>
          MOVE MESSAGE’BUF :=
              " No batch file was specified, so browse is not allowed.",2;

       << NO’NEXT’REC    >>
       MOVE MESSAGE’BUF :=
          " There are no more batch records.", 2;
       END;

   MSGLEN := TOS - @MESSAGE’BUF;

   VPUTWINDOW (COMAREA, MESSAGE’BUF, MSGLEN);

   END;   << ENTRY’ERROR >>
```

```
$PAGE"           ERROR"
<<****************************************************************>>
<<                                                                >>
<<                          ERROR                                 >>
<<                                                                >>
<<****************************************************************>>
PROCEDURE ERROR;
   BEGIN

   IF ERRORS THEN      << WILL ONLY HANDLE FIRST ERROR! >>
      RETURN;

   ERRORS := TRUE;

   MESSAGE'BUF := " ";
   VERRMSG (COMAREA, MESSAGE'BUF(1), MESSAGE'BUF'LEN, MSGLEN);
   MSGLEN := MSGLEN + 1;

   COM'STATUS := 0;
   VPUTWINDOW (COMAREA MESSAGE'BUF, MSGLEN);

   END; << ERROR >>
```

```
$PAGE "          ENTRY INITIALIZATION PROCEDURE"
<<****************************************************************>>
<<                                                                >>
<<                             INIT                               >>
<<                                                                >>
<<****************************************************************>>
PROCEDURE INIT;
   BEGIN

   EQUATE
         VERSIONS'DIFF = 70
        ,DIF'FF        = 73
        ,FILENAMELEN   = 36
        ,LANGID'LEN    = 17
        ;
   EQUATE
         BLANK'LINE    =  0
        ,GET'FF'NAME   =  1
        ,GET'BF'NAME   =  2
        ,DIF'FF'WARN   =  3
        ,VERS'DIF'WARN =  4
        ,Y'TO'CONT     =  5
        ,PRODUCT'ID    =  6
        ,GET'LANGID    =  7
        ,NOT'CONFIG    =  8
        ,NOT'INSTALL   =  9
        ;
   INTEGER
         INDEX
        ,READ'LEN
        ,LANGID
        ,VERROR
        ;
   LOGICAL
         CONTINUE
        ;
   LOGICAL ARRAY
         NLERROR(0:1)
        ;
   LOGICAL ARRAY
         LANGID'STR'L(0:9)
        ;
   BYTE ARRAY
         LANGID'STR(*) = LANGID'STR'L
        ;
   BYTE ARRAY
         FILENAME (0:FILENAMELEN)
        ;
   INTRINSIC
         TERMINATE
        ,QUIT
        ,PRINT
        ,READ
        ;
```

```
INTRINSIC
     NLGETLANG
     ,NLINFO
     ;

SUBROUTINE HANDLE'PROMPT'ERR (QUIT'NUM);
VALUE QUIT'NUM;
INTEGER QUIT'NUM;
     BEGIN
     MOVE MESSAGE'BUF := "Terminal access failed unexpectedly.", 2;
     MSGLEN := TOS - @MESSAGE'BUF;
     PRINT (MESSAGE'WBUF, -MSGLEN, 0);
     QUIT (QUIT'NUM);
     END;    << HANDLE'PROMPT'ERR >>

SUBROUTINE WRITE'MSG;
     BEGIN
     VERRMSG (COMAREA, MESSAGE'BUF, MESSAGE'BUF'LEN, MSGLEN);
     PRINT (MESSAGE'WBUF, -(MSGLEN), %60);
     IF <> THEN   << CANT WRITE TO PROMPT FILE! >>
        HANDLE'PROMPT'ERR (%60);
     END; << WRITE'MSG >>


SUBROUTINE PRINT'TO'TERM (MSG'NUM, CCTL);
VALUE MSG'NUM, CCTL;
INTEGER MSG'NUM;
LOGICAL CCTL;
     BEGIN

     CASE MSG'NUM OF
        BEGIN

        << 0, BLANK'LINE >>
        MOVE MESSAGE'BUF:=" ",2;

        << 1, FF'NAME'PROMPT >>
        MOVE MESSAGE'BUF:=" ENTER FORMS FILE NAME AND PRESS RETURN: ",
                             2;

        << 2, BF NAME PROMPT >>
        MOVE MESSAGE'BUF:=" ENTER BATCH FILE NAME AND PRESS RETURN: ",
                             2;

        << 3, DIFFERENT FF WARNING >>
        MOVE MESSAGE'BUF:=(" WARNING: A different forms file was used",
                             " to create this batch."),2;

        << 4, FF MOD WARN >>
        MOVE MESSAGE'BUF:=(" WARNING: Forms File was recompiled since",
                             " this batch was created."), 2;

        << 5, Y'TO'CONTINUE >>
        MOVE MESSAGE'BUF := (" Enter ""Y"" to continue: "), 2;

        << 6, PRODUCT'ID >>
        MOVE MESSAGE'BUF := ID'MSG, 2;
```

```
           << 7, GET'LANGID >>
           MOVE MESSAGE'BUF :=(" ENTER LANGUAGE ID NUMBER AND PRESS",
                                 "RETURN: "),2;

           << 8, NOT'CONFIG >>
           MOVE MESSAGE'BUF := " Specified language is not configured ",2;

           << 9, NOT'INSTALL >>
           MOVE MESSAGE'BUF := " Native language Software not installed",2
                                   ;

           END;   << CASE >>

       MSGLEN := TOS - @MESSAGE'BUF;
       PRINT (MESSAGE'WBUF, -MSGLEN, CCTL);
       IF <> THEN
           HANDLE'PROMPT'ERR (2);

       END;    << PRINT'TO'TERM >>
    INTEGER SUBROUTINE READ'FROM'TERM (READBUF, READLEN);
    VALUE READLEN;
    BYTE ARRAY READBUF;
    INTEGER READLEN;
       BEGIN

       << BLANK BUF FIRST >>
       READBUF := " ";
       MOVE READBUF (1) := READBUF (0), (READLEN-1);

       READ'FROM'TERM := READ (READBUF, -READLEN);
       IF <> THEN
           HANDLE'PROMPT'ERR (3);
       END;    << READ'FROM'TERM >>
```

```
$PAGE

   << INITIALIZE COMAREA; IS ALL 0'S TO START >>
   COM'LANGUAGE := SPL'LANG;
   COM'COMAREALEN := COMAREALEN;

   << SET COM'LABEL'OPTION TO 1 TO ENABLE FUNCTION KEY LABEL    >>
   << SUPPORT FOR TERMINALS SUPPORTING FUNCTION KEY LABELS      >>
   COM'LABEL'OPTION := 1;

   << Set form storage buffer size (2626 terminal only) to 4 >>
   COM'FORM'STOR'SIZE := 4;

   BATCH := TRUE; << INIT >>

   PRINT'TO'TERM (PRODUCT'ID, %60);    << ENTRY IDENTIFICATION >>

   WHILE TRUE DO
       BEGIN

       DO     << UNTIL COM'STATUS = 0 >>
          BEGIN
          COM'STATUS := 0;
          PRINT'TO'TERM (GET'FF'NAME, %320);
          READ'LEN := READ'FROM'TERM (FILENAME, FILENAMELEN);
          IF READ'LEN = 0 THEN   << ALL DONE >>
            TERMINATE;

          VOPENFORMF (COMAREA, FILENAME);
          IF COM'STATUS <> 0 THEN
             WRITE'MSG;   << WRITES VERRMSG >>
          END
       UNTIL COM'STATUS = 0;   << KEEP GOING TILL OK >>

       << NOW, OPEN BATCH FILE >>
       PRINT'TO'TERM (GET'BF'NAME, %320);
       READ'LEN := READ'FROM'TERM (FILENAME, FILENAMELEN);
       IF READ'LEN = 0 OR FILENAME = " " THEN   << NO BATCH FILE! >>
          BATCH := FALSE  << ALL OK >>
       ELSE
          BEGIN
          VOPENBATCH (COMAREA, FILENAME);
          IF COM'STATUS <> 0 THEN
             IF COM'STATUS = VERSIONS'DIFF OR
                 COM'STATUS = DIF'FF THEN
                 BEGIN
                 PRINT'TO'TERM ((IF COM'STATUS=DIF'FF THEN DIF'FF'WARN
                                 ELSE VERS'DIF'WARN), 0);
                 PRINT'TO'TERM (Y'TO'CONT, %320);
                 READ'LEN := READ'FROM'TERM (MESSAGE,BUF, 1);
                 IF READ'LEN > 0 THEN
                    IF READ'LEN=1 AND (MESSAGE'BUF = "Y" OR
                                       MESSAGE'BUF = "y") THEN
                       COM'STATUS := 0;    << GO AHEAD >>
                 END
                    ELSE    << IS REAL ERROR >>
                        WRITE'MSG;
            END;
       IF COM’STATUS = 0 THEN
            BEGIN

            VGETLANG( COMAREA, LANGID );
            IF COM’STATUS <> 0 THEN WRITE’MSG
            ELSE IF LANGID = INTERNATIONAL THEN BEGIN

            << IF INTERNATIONAL FORMS FILE PROMPT FOR LANGID >>

              CONTINUE := TRUE;
              LANGID := NLGETLANG( 1, NLERROR );
              IF NLERROR = 0 THEN BEGIN
                  VSETLANG( COMAREA LANGID, VERROR );
                  COM’STATUS := 0;
              END;

              WHILE CONTINUE DO BEGIN

                  PRINT’TO’TERM( BLANK’LINE, %40 );
                  PRINT’TO’TERM( GET’LANGID, %320 );
                  READ’LEN := READ’FROM’TERM( LANGID’STR, LANGID’LEN );
                  IF READ’LEN = 0 THEN CONTINUE := FALSE
                  ELSE BEGIN
                       LANGID’STR( READ’LEN ) :=" ";
                       NLINFO( 22, LANGID’STR’L, LANGID, NLERROR );
                       IF NLERROR = 0 THEN BEGIN
                          VSETLANG( COMAREA LANGID, VERROR );
                          IF VERROR = 0 AND COM’STATUS = 0 THEN
                             CONTINUE := FALSE;
                          IF COM’STATUS <> 0 THEN WRITE’MSG;
                          END
                       ELSE IF NLERROR = 1
                           THEN PRINT’TO’TERM(NOT’INSTALL,%40)
                           ELSE PRINT’TO’TERM(NOT’CONFIG,%40);
                  END;
              END;    << WHILE CONTINUE >>
            END;      << IF LANGID = INTERNATIONAL >>

       END;           << IF COM’STATUS = 0 >>

       << ALL OK HERE, SO OPEN TERMINAL >>
       IF NOT BATCH OR COM’STATUS = 0 THEN
            BEGIN

            << OPEN TERMINAL IN BLOCKMODE ... >>
            MOVE FILENAME := "A264X ";
            VOPENTERM (COMAREA, FILENAME);
            IF COM’STATUS <> 0 THEN
              BEGIN
              WRITE’MSG;
              QUIT (6);
              END;
        COM’TERMOPTIONS.(11:2) := 1;      << DONT HARD RESET TERM >>
        RETURN;   << ALL DONE INITIALIZING >>
        END
     ELSE   << IS NORMAL ERROR >>
        BEGIN
        COM’STATUS := 0;
        VCLOSEBATCH (COMAREA);
        VCLOSEFORMF (COMAREA);
        END;

     END;   << WHILE TRUE >>
END; << INIT >>
```

```
$PAGE "            EXIT"
<<****************************************************************>>
<<                                                                >>
<<                             EXIT                               >>
<<                                                                >>
<<****************************************************************>>
PROCEDURE EXIT;
   BEGIN

   BYTE ARRAY LOCAL'MESSAGE'BUF (0:80);
   INTEGER LOCAL'MSGLEN;

   INTRINSIC PRINT;
   SUBROUTINE PRINT'MSG;
        BEGIN
        VERRMSG (COMAREA, LOCAL'MESSAGE'BUF, MESSAGE'BUF'LEN,
                 LOCAL'MSGLEN);
        PRINT (LOCAL'MESSAGE'BUF, -LOCAL'MSGLEN, 0);
        COM'STATUS := 0;
        END;

   << FIRST, CLOSE TERMINAL >>
   COM'STATUS := 0;
   VCLOSETERM (COMAREA);
   IF COM'STATUS <> 0 THEN
        PRINT'MSG;

   << NOW, BATCH FILE >>
   IF BATCH THEN
        IF ERRORS THEN
          PRINT (MESSAGE'WBUF, -MSGLEN, 0) << MSG FROM COLLECT >>
        ELSE << OK TO GO AHEAD >>
          BEGIN
          VCLOSEBATCH (COMAREA);
          IF COM'STATUS <> 0 THEN
             PRINT'MSG;
          end
   else
        if errors then
          print (message'wbuf, -msglen, 0); << msg from collect >>

   << NOW, CLOSE FORMS FILE >>
    VCLOSEFORMF (COMAREA);
   IF COM'STATUS <> 0 THEN
        PRINT'MSG;

   END;    << EXIT >>
```

```
$PAGE "           BROWSE"
<<****************************************************************>>
<<                                                                >>
<<                        BROWSE                                  >>
<<                                                                >>
<<****************************************************************>>
PROCEDURE BROWSE;
   BEGIN

   EQUATE
        FORWARDS  =  1
       ,BACKWARDS = -1
       ;
   INTEGER
        PAGE'EJECT := %61
       ,UNDERLINE := 1
       ,DIRECTION
       ;
   DOUBLE
        LOCAL'COM'REC
       ;

    DO'BROWSE'LABELS;

   LOCAL'COM'REC := COM'RECNUM;
   COM'RECNUM := COM'RECNUM - 1D;
   DIRECTION := BACKWARDS;

   WHILE TRUE DO        << UNTIL EXIT OR COLLECTKEY >>
       BEGIN

       IF COM'NUMRECS = 0D THEN
          RETURN;

       IF COM'RECNUM = LAST'REC'NUM THEN
          BEGIN
          ENTRY'ERROR (NO'NEXT'RECS);
          COM'RECNUM := COM'RECNUM - 1D;
          DIRECTION := BACKWARDS;
          END;

       IF COM'RECNUM < 0D THEN
          BEGIN
          ENTRY'ERROR (NO'PREV'RECS);
          COM'RECNUM := 0D;
          DIRECTION := FORWARDS;
          END;

       VREADBATCH (COMAREA);
       CHECK'ERROR;

       IF COM'DELETEFLAG = FALSE THEN      << NOT DELETED >>
          BEGIN
       IF COM'RECNUM <> LOCAL'COM'REC OR COM'LASTKEY = REFRESHKEY THEN
             BEGIN
             IF DIRECTION = BACKWARDS OR COM'LASTKEY = REFRESHKEY THEN
                COM'REPEATOPT := COM'NFOPT := NORM
             ELSE   << MUST BE FORWARDS >>
```

```
         IF COM'CFNAME <> COM'NFNAME, (15) THEN
            COM'REPEATOPT := NORM;    << CLEAR SINCE NOT REPT >>

      IF COM'LASTKEY = REFRESHKEY THEN
         MOVE COM'NFNAME := "$REFRESH        ";

      VGETNEXTFORM (COMAREA);
      CHECK'ERROR;

      LOCAL'COM'REC := COM'RECNUM;
      END;

   IF NOT ERRORS THEN
      FORMAT'STATUS'LINE;

   DO << WHILE ERRORS >>
      BEGIN

      ERRORS := FALSE;

      VSHOWFORM (COMAREA);
      CHECK'ERROR

      COM'SHOWCONTROL := 0;     << RESET JUST IN CASE >>

      VREADFIELDS (COMAREA);
      CHECK'ERROR;

      if com'lastkey <> 0 then
        if com'term'type = 15 or       << HP3075 >>
           com'term'type = 16 then     << HP3076 >>
          if com'keyboard'type = 1 then << Numeric keyboard >>
            com'lastkey := com'lastkey - 16;

      IF NOT ERRORS THEN
         CASE COM'LASTKEY OF
            BEGIN

            << ENTERKEY: >>
               BEGIN
               DIRECTION := FORWARDS;

               VFIELDEDITS (COMAREA);
                CHECK'EDIT'ERROR;

               IF NOT ERRORS THEN
                  BEGIN
    $$VFINISHFORM (COMAREA);
    CHECK'EDIT'ERROR;

    IF COM'REPEATOPT=NOREPEAT AND COM'NFOPT <> NORM
    OR COM'REPEATOPT=REPEATAPP THEN
    BEGIN
    COM'SHOWCONTROL.(10:1) := 1;
    VSHOWFORM (COMAREA);
    COM'SHOWCONTROL. (10:1) := 0;
    CHECK'ERROR;
    END;
```

```
        IF NOT ERRORS THEN
           BEGIN
           VWRITEBATCH (COMAREA);
           CHECK'ERROR;

           IF NOT ERRORS THEN
           COM'RECNUM := COM'RECNUM+1D;
           END;

           END;

    END;

 << HEADKEY: >>
    BEGIN
    DIRECTION := FORWARDS;
    COM'RECNUM := 0D;
    COM'REPEATOPT := COM'NFOPT := NORM;
    END;

 << DELETEKEY: >>
    BEGIN
    DIRECTION := FORWARDS;

    COM'DELETEFLAG := TRUE;
    VWRITEBATCH (COMAREA);
    CHECK'ERROR;
    COM'DELETEFLAG := FALSE;
    IF NOT ERRORS THEN
           COM'RECNUM := COM'RECNUM + 1D;

    COM'REPEATOPT := COM'NFOPT := NORM;
    END;

 << PRINTKEY:  >>
    BEGIN
    VPRINTFORM (COMAREA, UNDERLIINE, PAGE'EJECT);
    CHECK'ERROR;
    END;
                   << REFRESHKEY: >>
                      ;

                   << PREVKEY: >>
                      BEGIN
                      DIRECTION := BACKWARDS;
                      COM'RECNUM := COM'RECNUM - 1D;
                      END;

                   << NEXTKEY: >>
                      BEGIN
                      DIRECTION := FORWARDS;
                      COM'RECNUM := COM'RECNUM + 1D;

                      IF COM'REPEATOPT=NOREPEAT AND COM'NFOPT <> NORM
                         OR COM'REPEATOPT=REPEATAPP THEN
                         BEGIN
                         COM'SHOWCONTROL.(10:1) := 1;
                         VSHOWFORM (COMAREA);
```

```
                          COM’SHOWCONTROL.(10:1) := 0;
                          CHECK’ERROR;
                          END;
                       END;

                    << COLLECTKEY: >>
                       RETURN;

                    << EXIT: >>
                       RETURN;

                    END;    << CASE >>

              END
          UNTIL NOT ERRORS AND COM’LASTKEY <> PRINTKEY;

          END     << IN NOT COM’DELETEFLAG >>
       ELSE       << REC WAS DELETED >>
          COM’RECNUM := IF DIRECTION = BACKWARDS THEN COM’RECNUM - 1D
                          ELSE COM’RECNUM + 1D;

       END; << WHILE TRUE DO >>

    END;   << BROWSE >>
```

```
$PAGE "           COLLECT"

<<*****************************************************************>>
<<                                                                 >>
<<                           COLLECT                               >>
<<                                                                 >>
<<*****************************************************************>>
PROCEDURE COLLECT;
   BEGIN

   LOGICAL
        FIRST’TIME := TRUE
        ;
   BYTE ARRAY
        SAVED’FORM’NAME (0:NAMELEN-1)
        ;

   DO’COLLECT’LABELS;

   COM’MODE := COLLECT’MODE;
   COM’DELETEFLAG := FALSE;

   DO    << UNTIL COM’NFNAME <> EXIT AND COM’DO <> NORM >>
        BEGIN

        IF COM’LASTKEY=ENTERKEY OR COM’LASTKEY=NEXTKEY THEN
          IF COM’REPEATOPT=NOREPEAT AND COM’NFOPT <> NORM OR
              COM’REPEATOPT=REPEATAPP THEN
              BEGIN

              COM’SHOWCONTROL.(10:1) := 1;
              << TO SUPPRESS KEYBOARD ENABLE >>
              VSHOWFORM (COMAREA);
              COM’SHOWCONTROL.(10:1) := 0;
              CHECK’ERROR;
              END;

        VGETNEXTFORM (COMAREA);
        IF FIRST’TIME AND COM’STATUS <> 0 THEN    << IS FIRST TIME >>
          BEGIN
         VERRMSG (COMAREA, MESSAGE’BUF, MESSAGE’BUF’LEN, MSGLEN);
          ERRORS := TRUE;     << DONT WANT TO CLOSE BATCH IF ERROR!  >>
          RETURN;
          END;

        CHECK’ERROR;
        FIRST’TIME := FALSE;

        VINITFORM (COMAREA);
        CHECK’EDIT’ERROR;

        IF NOT ERRORS THEN
          FORMAT’STATUS’LINE;

        DO   << WHILE ERRORS >>

          BEGIN
          ERRORS := FALSE;

          VSHOWFORM (COMAREA);
          CHECK'ERROR;

          COM'SHOWCONTROL := 0;        << CLEAR >>

          IF COM'DBUFLEN <= 0 AND      << DONT READ!!! >>
              COM'REPEATOPT=NOREPEAT AND COM'NFOPT <> NORM THEN
              BEGIN
              IF NOT ERRORS AND BATCH THEN
                 BEGIN
                 VWRITEBATCH (COMAREA).
                 CHECK'ERROR;

                 IF NOT ERRORS THEN
                     BEGIN
                     COM'RECNUM := COM'RECNUM + 1D;
                     IF (COM'RECNUM MOD DOUBLE(PARMVAL) = 0D) THEN
                        VPOSTBATCH (COMAREA);
                     END;
                 END;
              END
          ELSE     << IS NORMAL FORM >>
              BEGIN
              VREADFIELDS (COMAREA);
              CHECK'ERROR;

              if com'lastkey <> 0 then
                if com'term'type = 15 or          << HP3075 >>
                   com'term'type = 16 then        << Hp3076 >>
                  if com'keyboard'type = 1 then << Numeric keyboard >>
                     com'lastkey := com'lastkey - 16;

              IF NOT ERRORS THEN
                 CASE COM'LASTKEY OF
                     BEGIN

                      << ENTERKEY: >>
                         BEGIN
                         VFIELDEDITS (COMAREA);
                         CHECK'EDIT'ERROR;

                         IF NOT ERRORS THEN
                            BEGIN
                            VFINISHFORM (COMAREA);
                            CHECK'EDIT'ERROR;

                            IF NOT ERRORS AND BATCH THEN
                               BEGIN
                               VWRITEBATCH (COMAREA);
                               CHECK'ERROR;

                            IF NOT ERRORS THEN
                   BEGIN
                   COM'RECNUM := COM'RECNUM + 1D;
                   IF (COM'RECNUM MOD DOUBLE(PARMVAL) = 0D) THEN
                      VPOSTBATCH (COMAREA);
```

```
END;

          END;

       END;

    END;

 << HEADKEY: >>
    BEGIN
    COM'REPEATOPT := NORM;
    COM'NFOPT    := NORM;
    MOVE COM'NFNAME := "$HEAD         ";
    END;

 << DELETEKEY: >>
    ENTRY'ERROR (DELETE'NOT'DEFINED);

 << PRINTKEY:    >>
    BEGIN
    VPRINTFORM (COMAREA, UNDERLINE, PAGE'EJECT);
    CHECK'ERROR;
    END;

 << REFRESHKEY: >>
    MOVE COM'NFNAME := "$REFRESH       ";

 << PREVKEY: >>
    ENTRY'ERROR (PREV'NOT'ALLOWED);

 << NEXTKEY: >>
    BEGIN
    IF COM'REPEATOPT = NORM THEN
       ENTRY'ERROR (NOT'REPEATING)
    ELSE
       COM'REPEATOPT := NORM;
    END;

 << BROWSEKEY:   >>
    BEGIN
    IF NOT BATCH THEN
       ENTRY'ERROR (NO'BATCH)
    ELSE
       IF COM'NUMRECS = 0D THEN
          ENTRY'ERROR (NO'BATCH'RECS)
       ELSE
          BEGIN

          LAST'REC'NUM := COM'RECNUM;
          MOVE SAVED'FORM'NAME := COM'CFNAME,(NAMELEN);
          COM'MODE := BROWSE,MODE;
          COM'REPEATOPT := COM'NFOPT := NORM;

          COM'SHOWCONTROL.(14:1):=1;
          BROWSE;
          COM'SHOWCONTROL.(14:1):=0;
          COM'MODE := COLLECT'MODE;
          MOVE COM'NFNAME := SAVED'FORM'NAME,(NAMELEN);
```

```
                            COM’RECNUM := LAST’REC’NUM;
                            COM’REPEATOPT := COM’NFOPT := NORM;
                            COM’DELETEFLAG := FALSE; << IF NO RECS >>

                            IF COM’LASTKEY = EXITKEY THEN
                               BEGIN
                               MOVE COM’CFNAME :=
                               SAVED’FORM’NAME,(NAMELEN);
                               RETURN;
                               END;

                            DO’COLLECT’LABELS;

                            END;

                         END;     << BROWSEKEY >>

                   << EXIT: >>
                      RETURN;

                   END; << CASE COM’LASTKEY >>

             END;   << IS COM’DBUFLEN > 0? >>
          END
       UNTIL NOT ERRORS AND COM’LASTKEY <> PRINTKEY;

       END
    UNTIL COM’NFNAME = "$END           " AND
          COM’REPEATOPT = NORM;

    END;   << COLLECT >>
```

```
$PAGE "          ENTRY OUTER BLOCK"
<<****************************************************************>>
<<                                                                >>
<<                      OUTER BLOCK                               >>
<<                                                                >>
<<****************************************************************>>

INTRINSIC PRINT;    << FOR ID MESSAGE >>

<< FOR INTERNAL TESTING ONLY >>

INIT;

COLLECT;

EXIT;

END.
```

COBOL

```
$CONTROL LIST, MAP, VERBS

 IDENTIFICATION DIVISION.

 PROGRAM-ID. COBOL-EXAMPLE.

*****
***** This application collects employee payroll deduction
***** transactions and places the edited transactions into
***** a file.
*****
***** For this application:          Enter key = edit and file
*****                                transaction;
*****
*****                                      f8 = exit application;
*****
*****                        all other f keys = redo transaction.
*****
***** Each transaction entered by the operator is subjected to the
***** data edits embedded within the input form.
*****
***** The application continues to collect transactions until either
***** the operator signals to exit or a system error is detected.
*****

 ENVIRONMENT DIVISION.

 INPUT-OUTPUT SECTION.

 FILE-CONTROL.

 SELECT TXN-ENTRY ASSIGN TO "PAYTXN".

 DATA DIVISION.

 FILE SECTION.

 FD TXN-ENTRY
     RECORD CONTAINS 200 CHARACTERS
     DATA RECORDS ARE TXN-REC.

 01 TXN-REC.
     05 FILLER                   PIC X(200).
 WORKING-STORAGE SECTION.


 01 COMAREA.
     05  CSTATUS                 PIC S9(4) COMP VALUE 0.
     05  LANGUAGE                PIC S9(4) COMP VALUE 0.
     05  COMAREALEN              PIC S9(4) COMP VALUE 0.
     05  USERBUFLEN              PIC S9(4) COMP VALUE 0.
     05  CMODE                   PIC S9(4) COMP VALUE 0.
     05  LASTKEY                 PIC S9(4) COMP VALUE 0.
     05  NUMERRS                 PIC S9(4) COMP VALUE 0.
```

```
    05   WINDOWENH           PIC S9(4) COMP VALUE 0.
    05   MULTIUSAGE          PIC S9(4) COMP VALUE 0.
    05   LABELOPTIONS        PIC S9(4) COMP VALUE 0.
    05   CFNAME              PIC X(16) VALUE SPACES.
    05   NFNAME              PIC X(16) VALUE SPACES.
    05   REPEATAPP           PIC S9(4) COMP VALUE 0.
    05   FREEZEAPP           PIC S9(4) COMP VALUE 0.
    05   CFNUMLINES          PIC S9(4) COMP VALUE 0.
    05   DBUFLEN             PIC S9(4) COMP VALUE 0.
    05   FILLER              PIC S9(4) COMP VALUE 0.
    05   LOOKAHEAD           PIC S9(4) COMP VALUE 0.
    05   DELETEFLAG          PIC S9(4) COMP VALUE 0.
    05   SHOWCONTROL         PIC S9(4) COMP VALUE 0.
    05   FILLER              PIC S9(4) COMP VALUE 0.
    05   PRINTFILENUM        PIC S9(4) COMP VALUE 0.
    05   FILERRNUM           PIC S9(4) COMP VALUE 0.
    05   ERRFILENUM          PIC S9(4) COMP VALUE 0.
    05   FORMSTORESIZE       PIC S9(4) COMP VALUE 0.
    05   FILLER              PIC S9(4) COMP VALUE 0.
    05   FILLER              PIC S9(4) COMP VALUE 0.
    05   FILLER              PIC S9(4) COMP VALUE 0.
    05   NUMRECS             PIC S9(8) COMP VALUE 0.
    05   RECNUM              PIC S9(8) COMP VALUE 0.
    05   FILLER              PIC S9(4) COMP VALUE 0.
    05   FILLER              PIC S9(4) COMP VALUE 0.
    05   TERMFILEN           PIC S9(4) COMP VALUE 0.
    05   FILLER              PIC S9(4) COMP VALUE 0.
    05   FILLER              PIC S9(4) COMP VALUE 0.
    05   FILLER              PIC S9(4) COMP VALUE 0.
    05   FILLER              PIC S9(4) COMP VALUE 0.
    05   FILLER              PIC S9(4) COMP VALUE 0.
    05   RETRIES             PIC S9(4) COMP VALUE 0.
    05   TERMOPTIONS         PIC S9(4) COMP VALUE 0.
    05   ENVIRON             PIC S9(4) COMP VALUE 0.
    05   USERTIME            PIC S9(4) COMP VALUE 0.
    05   IDENTIFIER          PIC S9(4) COMP VALUE 0.
    05   LABELINFO           PIC S9(4) COMP VALUE 0.
01   FIELDINFO.

    05   NUM-ENTRIES         PIC S9(4) COMP.
    05   ENTRY-LEN           PIC S9(4) COMP.
    05   FORM-NAME           PIC X(16).
    05   FIELD-ENTRY    OCCURS 3 TIMES.
         10 FIELD-NAME       PIC X(16).
         10 SCREEN-ORDER     PIC S9(4) COMP.

01   FIELDSPECS.
    05   SPEC-ENTRY     OCCURS  3 TIMES.
         10  FIELD-ID        PIC S9(4) COMP.
         10 CHANGE-TYPE      PIC S9(4) COMP.
         10  CHANGE-SPEC     PIC X(4).

01   DATABUF                 PIC X(200).

  01  DATABUFLEN              PIC S9(4) COMP.

  01  DONE-WITH-TRANSACTIONS  PIC X.
  01  ERROR-LOCATION          PIC X(70).

  01  FILENAME                PIC X(86).

  01  FOUND-DATA-ERRORS       PIC X.

  01  INFOBUFLEN              PIC S9(4) COMP.

  01  MSGBUF                  PIC X(150).

  01  MSGBUFLEN               PIC S9(4) COMP.

  01  ERRMSGLEN               PIC S9(4) COMP.

  01  NBR-TXN-COLLECTED       PIC 9(4).

  01  NUMSPECS                PIC S9(4) COMP.

  01  STOP-NOW                PIC X.
  PROCEDURE DIVISION.

  A-000-START-PROGRAM.

      MOVE "N" TO STOP-NOW
                  DONE-WITH-TRANSACTIONS.

      MOVE ZERO TO NBR-TXN-COLLECTED.

      PERFORM A-100-SETUP-FOR-WORK.

      PERFORM A-500-COLLECT-TRANSACTIONS
         UNTIL STOP-NOW               = "Y"
            OR DONE-WITH-TRANSACTIONS = "Y".

      PERFORM A-900-CLEANUP-AFTER-WORK.

      DISPLAY " ".
      DISPLAY "Deduction transactions collected this session = "
               NBR-TXN-COLLECTED.

      IF STOP-NOW = "Y"
         PERFORM Z-900-DISPLAY-SYSTEM-ERROR.

      STOP RUN.
  A-100-SETUP-FOR-WORK.

 *****
 ***** Finish Comarea initialization.
 *****

***** (Note Comarea value clauses.)
*****

*****
*****       Set Language for COBOL.
*****

     MOVE ZERO TO LANGUAGE OF COMAREA.
```

```
*****
*****       Set Comarealen to 60 words (120 bytes).
*****

     MOVE 60 TO COMAREALEN OF COMAREA.

*****
*****       Activate function key labeling.
*****

     MOVE 1 TO LABELOPTIONS OF COMAREA.

*****
*****       Disable form background loading on Vreadfields.
*****

     MOVE ZERO TO LOOKAHEAD OF COMAREA.

*****
*****       Set size of local form storage directory.
*****

     MOVE 4 TO FORMSTORESIZE OF COMAREA.

*****
***** Open the Transaction File
*****

     OPEN OUTPUT TXN-ENTRY.

*****
***** Open the Forms File.
*****

     MOVE "PAYROLL.WORK.ADMIN" TO FILENAME.

     CALL "VOPENFORMF" USING COMAREA
                             FILENAME.

     IF CSTATUS OF COMAREA NOT = 0
        MOVE "Y" TO STOP-NOW
        MOVE
          "**** Paragraph: A-100-SETUP-FOR-WORK - Forms File Open"
            TO ERROR-LOCATION
        PERFORM  Z-100-GET-ERROR-MESSAGE.
     IF STOP-NOW NOT = "Y"

*****
***** Open the Terminal.
*****

         MOVE "HPTERM" TO FILENAME

         CALL "VOPENTERM" USING COMAREA
                                FILENAME

         IF CSTATUS OF COMAREA NOT = 0
```

```
            MOVE "Y" TO STOP-NOW
            MOVE
             "**** Paragraph: A-100-SETUP-FOR-WORK - Terminal Setu
                "p"
              TO ERROR-LOCATION
            PERFORM Z-100-GET-ERROR-MESSAGE.
     IF STOP-NOW NOT = "Y"

*****
***** Translate field names to screen orders.
*****
*****    Three of the fields in the form used by this
*****    application need to be toggled from "display
*****    only" to "input allowed". In order to do this,
*****    we first translate field names to screen orders.
*****

         MOVE 3 TO NUM-ENTRIES OF FIELDINFO

         MOVE 9 TO ENTRY-LEN OF FIELDINFO

         MOVE "DEDUCTION" TO FORM-NAME OF FIELDINFO

*****
***** The value 8224, which is moved to Screen Order in
***** the following statements is equal to two ASCII blanks.
*****
         MOVE "BADGE_NUMBER" TO FIELD-NAME
            OF FIELD-ENTRY (1)
         MOVE 8224 TO SCREEN-ORDER
            OF FIELD-ENTRY (1)

         MOVE "LAST-NAME" TO FIELD-NAME
            OF FIELD-ENTRY (2)
         MOVE 8224 TO SCREEN-ORDER
            OF FIELD-ENTRY (2)

         MOVE "SUR NAME" TO FIELD-NAME
            OF FIELD-ENTRY (3)
         MOVE 8224 TO SCREEN-ORDER
            OF FIELD-ENTRY (3)
*****
***** Now determine the length of the entire Fieldinfo
***** Buffer.
*****

                MULTIPLY NUM-ENTRIES OF FIELDINFO
                     BY ENTRY-LEN OF FIELDINFO
                     GIVING INFOBUFLEN

                ADD 10 TO INFOBUFLEN

                CALL "VGETFIELDINFO" USING COMAREA
                                           FIELDINFO
                                           INFOBUFLEN

                IF CSTATUS OF COMAREA NOT = 0
                     MOVE "Y" TO STOP-NOW
```

```
                    MOVE
                    "**** Paragraph: A-100-SETUP-FOR-WORK - Field Informa
                          "tion Retrieval"
                        TO ERROR-LOCATION
                    PERFORM Z-100-GET-ERROR-MESSAGE.
 A-500-COLLECT-TRANSACTIONS.

*****
***** Setup for and get transaction data entry form.
*****

        MOVE ZERO TO REPEATAPP OF COMAREA
                     FREEZEAPP OF COMAREA.

        MOVE "DEDUCTION" TO NFNAME OF COMAREA.

        CALL "VGETNEXTFORM" USING COMAREA.

        IF CSTATUS OF COMAREA NOT = 0
           MOVE "Y" TO STOP-NOW
           MOVE
            "***** Paragraph:   A-500-COLLECTION-TRANSACTIONS - Form R
               "etrieval"
              TO ERROR-LOCATION
           PERFORM Z-100-GET-ERROR-MESSAGE.

        IF STOP-NOW NOT = "Y"

*****
***** Toggle three fields in form to "input allowed".
*****
*****            Screen order is indicated to field change
*****            intrinsic as negative number.
*****
*****            Change field type is indicated by a 5.
*****
*****            "Input allowed" is indicated by an "O"
*****      (for input/output).
*****

        MULTIPLY SCREEN-ORDER OF FIELD-ENTRY (1)

            BY -1
            GIVING FIELD-ID OF SPEC-ENTRY (1)
        MOVE 5 TO CHANGE-TYPE OF SPEC-ENTRY (1)
        MOVE "O" TO CHANGE-SPEC OF SPEC-ENTRY (1)

        MULTIPLY SCREEN-ORDER OF FIELD-ENTRY (2)
            BY -1
            GIVING FIELD-ID OF SPEC-ENTRY (2)
        MOVE 5 TO CHANGE-TYPE OF SPEC-ENTRY (2)
        MOVE "O" TO CHANGE-SPEC OF SPEC-ENTRY (2)

        MULTIPLY SCREEN-ORDER OF FIELD-ENTRY (3)
           BY -1
           GIVING FIELD-ID OF SPEC-ENTRY (3)
        MOVE 5 TO CHANGE-TYPE OF SPEC-ENTRY (3)
        MOVE "O" TO CHANGE-SPEC OF SPEC-ENTRY (3)
```

```
         MOVE 3 TO NUMSPECS
         CALL "VCHANGEFIELD" USTNG COMAREA
                                   FIELDSPES
                                   NUMSPECS

         IF CSTATUS OF COMAREA NOT = 0
             MOVE "Y" TO STOP-NOW
             MOVE
              "**** Paragraph: A-500-COLLECT-TRANSACTIONS - Field
                 "Type Updates"
                TO ERROR-LOCATION
             PERFORM Z-100-GET-ERROR-MESSAGE.

      IF STOP-NOW NOT =  "Y"

*****
***** Load window message.
*****
         MOVE 79 TO MSGBUFLEN

         MOVE
           "Fill in Deduction Transaction according to worksheet."
                TO MSGBUF

         CALL "VPUTWINDOW" USING COMAREA
                                 MSGBUF
                                 MSGUFLEN

         IF CSTATUS OF COMAREA NOT = 0
             MOVE "Y" TO STOP-NOW
             MOVE
              "**** Paragraph: A-500-COLLECT-TRANSACTIONS - Window
<…sc><…x>

<ex><esc>
                                "Load"
                          TO ERROR-LOCATION
                    PERFORM Z-100-GET-ERROR-MESSAGE.

               IF STOP-NOW NOT = "Y"

*****
***** Initialize form.
*****

               CALL "VINITFORM" USING COMAREA

               IF CSTATUS OF COMAREA NOT = 0
                  MOVE "Y" TO STOP-NOW
                  MOVE
                   "**** Paragraph: A-500-COLLECT-TRANSACTIONS - Form I
                      "nit "
                     TO ERROR-LOCATION
                  PERFORM Z-100-GET-ERROR-MESSAGE.
           IF STOP-NOW NOT = "Y"

*****
```

```
***** Show form.
*****

               CALL "VSHOWFORM" USING COMAREA

               IF CSTATUS OF COMAREA NOT = 0
                  MOVE "Y" TO STOP-NOW
                  MOVE
                   "**** Paragraph: A-500-COLLECT-TRANSACTIONS - Form D
                      "isplay"
                     TO ERROR-LOCATION
                  PERFORM Z-100-GET-ERROR-MESSAGE.

           IF STOP-NOW NOT = "Y"

*****
***** Setup and loop on transaction until it can be filed.
*****


               MOVE "Y" TO FOUND-DATA-ERRORS


               PERFORM        B-100-READ-EDIT-AND-FILE
                  UNTIL        FOUND-DATA-ERRORS = "N"
                     OR        STOP-NOW = "Y"
                     OR        DONE-WITH-TRANSACTIONS = "Y".
        B-100-READ-EDIT-AND-FILE.

*****
***** Read form.
*****

      CALL "VREADFIELDS" USING COMAREA.
         IF CSTATUS OF COMAREA NOT = 0
            MOVE "Y" TO STOP-NOW
            MOVE
             "**** Paragraph: B-100-READ-EDIT-AND-FILE - Terminal Rea
                "d"
               TO ERROR-LOCATION
            PERFORM Z-100-GET-ERROR-MESSAGE.

         IF STOP-NOW NOT = "Y"

*****
***** Determine if operator wants to stop transaction collection.
*****

                IF LASTKEY OF COMAREA = 1
                   MOVE "Y" TO DONE-WITH-TRANSACTIONS.

               IF STOP-NOW NOT = "Y"
                AND DONE-WITH-TRANSACTIONS NOT = "Y"

*****
***** Edit data read from terminal
*****
                CALL "VFIELDEDITS" USING COMAREA
                IF CSTATUS OF COMAREA NOT = 0
                   MOVE "Y" TO STOP-NOW
                   MOVE
                    "**** Paragraph: B-100-READ-EDIT-AND-FILE - Data Edit
                       " "
                      TO ERROR-LOCATION
                   PERFORM Z-100-GET-ERROR-MESSAGE

                ELSE

*****
***** Determine if edit errors detected.
*****

                     IF NUMERRS OF COMAREA < 1
                        MOVE "N" TO FOUND-DATA-ERRORS.
          IF        STOP-NOW NOT = "Y"
           AND DONE-WITH-TRANSACTIONS NOT = "Y"
           AND FOUND-DATA-ERRORS NOT = "Y"



*****
***** Finish form data.
*****
        CALL "VFINISHFORM" USING COMAREA

        IF CSTATUS OF COMAREA NOT = 0
           MOVE "Y" TO STOP-NOW


                   MOVE
                    "**** Paragraph: B-100-READ-EDIT-AND-FILE - Data Fini
                       "shing"
                      TO ERROR-LOCATION
                   PERFORM Z-100-GET-ERROR-MESSAGE

                ELSE

*****
***** Determine if data finishing errors detected.
*****

                    IF NUMERRS OF COMAREA > 0
                       MOVE "Y" TO FOUND-DATA-ERRORS.
        IF         STOP-NOW NOT = "Y"
          AND DONE-WITH-TRANSACTIONS NOT = "Y"

*****
***** Do we have a transaction that can be filed?
*****

        IF FOUND-DATA-ERRORS NOT = "Y"

           IF LASTKEY OF COMAREA = 0

*****
***** Get transaction from form and file it.
```

```
*****

              MOVE SPACES TO DATABUF

              MOVE 200 TO DATABUFLEN

              CALL "VGETBUFFER" USING COMAREA
                                      DATABUF
                                      DATABUFLEN

              IF CSTATUS OF COMAREA NOT = 0

                 MOVE "Y" TO STOP-NOW
                 MOVE
                  "**** Paragraph: B-100-READ-EDIT-AND-FILE - Dat
                     "a Get"
                    TO ERROR-LOCATION
                 PERFORM Z-100-GET-ERROR-MESSAGE

              ELSE

                 WRITE TXN-REC FROM DATABUF

                 ADD 1 TO NBR-TXN-COLLECTED.
IF         STOP-NOW NOT = "Y"
 AND DONE-WITH-TRANSACTIONS NOT = "Y"

*****
***** Do we need to prompt the operator to correct errors?
*****

           IF FOUND-DATA-ERRORS = "Y"

               IF LASTKEY OF COMAREA = 0

                   PERFORM B-200-PROMPT-OPERATOR.

        IF   STOP-NOW NOT = "Y"
        AND DONE-WITH-TRANSACTIONS NOT = "Y"

*****
***** Do we need to refresh the display?
*****

           IF FOUND-DATA-ERRORS = "Y"

               IF LASTKEY OF COMAREA NOT = 0

*****
***** The operator pressed some key other than Enter
***** or Exit so clear data error flag to break loop.
*****

                   MOVE "N" TO FOUND-DATA-ERRORS.
 B-200-PROMPT-OPERATOR.


*****
```

```
***** Get message text associated with first field flagged
***** with a data error.
*****

        PERFORM Z-100-GET-ERROR-MESSAGE.

        CALL "VPUTWINDOW" USING COMAREA
                                MSGBUF
                                ERRMSGLEN.

        IF CSTATUS OF COMAREA NOT = 0
           MOVE "Y" TO STOP-NOW
           MOVE
            "**** Paragraph: B-200-PROMPT-OPERATOR - Window Load"
               TO ERROR-LOCATION
           PERFORM      Z-100-GET-ERROR-MESSAGE.

        IF STOP-NOW NOT = "Y"

*****
***** Display highlighted form and updated window message.
*****

         CALL "VSHOWFORM" USING COMAREA.
                   IF CSTATUS OF COMAREA NOT = 0
                      MOVE "Y" TO STOP-NOW
                      MOVE
                        "**** Paragraph: B-200-PROMPT-OPERATOR - Display Upd
                           "ates"
                           TO ERROR-LOCATION
                      PERFORM Z-100-GET-ERROR-MESSAGE.

  A-900-CLEANUP-AFTER-WORK.

*****
***** Note that this paragraph unconditionally attempts to
***** close the Forms File and Terminal.
*****

            CLOSE TXN-ENTRY.

            MOVE 0 to CSTATUS OF COMAREA.

            CALL "VCLOSEFORMF" USING COMAREA.

            MOVE 0 to CSTATUS OF COMAREA.

            CALL "VCLOSETERM" USING COMAREA.


  Z-100-GET-ERROR-MESSAGE.

            MOVE SPACES TO MSGBUF.
            MOVE 150 TO MSGBUFLEN.

            CALL "VERRMSG" USING COMAREA
                                 MSGBUF
                                 MSGBUFLEN
```

```
                         ERRMSGLEN

Z-900-DISPLAY-SYSTEM-ERROR.

          DISPLAY "**** Transaction entry facility detected system erro
                    "r at:".
          DISPLAY ERROR-LOCATION.
          DISPLAY "**** The error message returned is:".
          DISPLAY "**** "
                  MSGBUF.
```

FORTRAN 77

```
$CONTROL list on, tables on
!
!  This application collects employee payroll deduction
!  transactions and places the edited transactions into
!  a file.
!
!  For this application:          Enter key = edit and file
!                                 transaction;
!

!                                         f8 = exit application;
!
!                          all other f keys = redo transaction.
!
!  Each transaction entered by the operator is subjected to the
!  data edits embedded within the input form.
!
!  The application continues to collect transactions until either
!  the operator signals to exit or a system error is detected.
!



$TITLE '                 Main Program'
!****************************************************************!
!                                                                !
!                      Main Program                              !
!                                                                !
!****************************************************************!
!
      PROGRAM FTNEXMP
!
      IMPLICIT NONE
!
      COMMON /COM01/ COMAREA
      COMMON /COMll/ STOP_NOW
      COMMON /COM12/ DONE WITH TXNS
      COMMON /COM13/ NBR_TXN_COLLECTED
      COMMON /COM21/ FIEEDINFO
      COMMON /COM22/ INFOBUFLEN
      COMMON /COM81/ MSGBUF
      COMMON /COM82/ MSGBUFLEN
      COMMON /COM83/ ERRMSGLEN
      COMMON /COM91/ ERROR_LOCATION
!
      INTEGER*2     COMAREA(60)
      INTEGER*2     STOP_NOW
      INTEGER*2     DONE_WITH_TXNS
      INTEGER*2     NBR_TXN_COLLECTED
      INTEGER*2     FIELDINFO(37)
      INTEGER*2     INFOBUFLEN
      CHARACTER*150 MSGBUF
     INTEGER*2        MSGBUFLEN
     INTEGER*2        ERRMSGLEN
     CHARACTER*70     ERROR_LOCATION
!
     STOP_ NOW = 0
     DONE_ WITH_TXNS = 0
!
     NBR_TXN_COLLECTED = 0
!
     CALL SETUP_FOR_WORK
!
     DO WHILE (STOP NOW.EQ.0
     +   .AND.DONE_WITH_TXNS.EQ.0)
        CALL COLLECT_TXNS
     END DO
!
     CALL CLEANUP_AFTER _ WORK
!
     PRINT *,
    + "Deduction transactions collected this session =",
    + NBR_TXN_COLLECTED
!
     IF (STOP NOW.EQ.1) THEN
        CALL DISPLAY_SYSTEM_ERROR
     END IF
!
     STOP
     END
```

```
$TITLE '                Setup For Work'

!***************************************************************!
!                                                               !
!                      Setup For Work                           !
!                                                               !
!***************************************************************!
!
       SUBROUTINE SETUP-FOR -WORK
!
       IMPLICIT NONE
!
       COMMON /COM01/ COMAREA
       COMMON /COMll/ STOP_NOW
       COMMON /COM21/ FIELDINFO
       COMMON /COM22/ INFOBUFLEN
       COMMON /COM81/ MSGBUF
       COMMON /COM82/ MSGBUFLEN
       COMMON /COM83/ ERRMSGLEN
       COMMON /COM91/ ERROR_LOCATION
!
       SYSTEM INTRINSIC VOPENFORMF,
      +                 VOPENTERM,
      +                 VGETFIELDINFO
!
       INTEGER*2      COMAREA(60)
       INTEGER*2      CSTATUS
       INTEGER*2      LANGUAGE
       INTEGER*2      COMAREALEN
       INTEGER*2      LABELOPTIONS
       INTEGER*2      LOOKAHEAD
       INTEGER*2      FORMSTORESIZE
       EQUIVALENCE  (COMAREA(1),  CSTATUS),
      +             (COMAREA(2),  LANGUAGE),
      +             (COMAREA(3),  COMAREALEN),
      +             (COMAREA(10), LABELOPTIONS),
      +             (COMAREA(32), LOOKAHEAD),
      +             (COMAREA(39), FORMSTORESIZE)
       INTEGER*2      STOP_NOW
       INTEGER*2      FIELDINFO(37)
       INTEGER*2      NUM_ENTRIES
       INTEGER*2      ENTRY_LEN
       CHARACTER*16   FORM_NAME
       EQUIVALENCE  (FIELDINFO(1),   NUM_ENTRIES),
      +             (FIELDINFO(2),  ENTRY_LEN),
      +             (FIELDINFO(3),  FORM_NAME)
            CHARACTER*18 FIELD_NAME (1,3)
            EQUIVALENCE (FIELDYNFO(11), FIELD_NAME)
       INTEGER*2      INFOBUFLEN
       CHARACTER*150 MSGBUF
       INTEGER*2      MSGBUFLEN
       INTEGER*2      ERRMSGLEN
       CHARACTER*70   ERROR_LOCATION
       INTEGER*2      ARRAY_INDEX
       CHARACTER*86   FILENAME
```

```
!
! Init Comarea to all zeros.
!
             ARRAY_INDEX = 1
             DO WHILE (ARRAY_INDEX.LE.60)
                COMAREA(ARRAY_INDEX) = 0
                ARRAY_INDEX = ARRAY_INDEX + 1
             END DO
!
! Set Language for FORTRAN-77.
!
             LANGUAGE = 5
!
! Set Comarealen for 60 words (120 bytes).
!
             COMAREALEN = 60
!
! Activate function key labeling.
!
             LABELOPTIONS = 1
!
! Disable form background loading on Vreadfields.
!
             LOOKAHEAD = 0
!
! Set size of local form storage directory.
!
             FORMSTORESIZE = 4
!
! Open the Transaction File:
!
             OPEN (UNIT   = 10,
            +      ENTITY = 'PAYTXN',
            +      ACCESS = 'DIRECT',
            +      RECL   = 200,
            +      FORM   = 'UNFORMATTED',
            +      STATUS = 'NEW',
            +      ERR    = 110)
!
             GOTO 120
!
  110 STOP_NOW = 1
             ERROR_LOCATION =
           + "**** Routine: Setup For Work - Open Transaction File"
             MSGBUF =
           + "**** File open failed!"
!
! Open the Forms File.
!
  120 IF (STOP NOW.EQ.0) THEN
             FILENIME = "PAYROLL.WORK.ADMIN"
!
          CALL VOPENFORMF (COMAREA,
              +            FILENAME)
!
               IF (CSTATUS.NE.0) THEN
                     STOP_NOW = 1
                     ERROR_LOCATION =
        +              "**** Routine: Setup For Work - Forms File Open"
                      CALL GET_ERROR_MESSAGE
                  END IF
              END IF
!
! Open the Terminal.
!
              IF (STOP NOW.EQ.0) THEN
                  FILENAME = "HPTERM"
!
                  CALL VOPENTERM (COMAREA,
        +                         FILENAME)
!
                  IF (CSTATUS.NE.0) THEN
                      STOP_NOW = 1
                      ERROR_LOCATION =

        +              "**** Routine: Setup For Work - Terminal Setup"
                      CALL GET_ERROR_MESSAGE
                  END IF
              END IF
!
! Translate field names to screen orders.
!
!       Three of the fields in the form used by this
!       application need to be toggled from "display
!       only" to "input allowed". In order to do this,
!       we first translate field names to screen orders.
!
              IF (STOP_NOW.EQ.0) THEN
!
                  NUM_ENTRIES = 3
                  ENTRY_LEN   = 9
                  FORM_NAME = "DEDUCTION"
!
! Note that because the FIELD_NAME element is defined as 18
! characters long, each occurrence of FIELD_NAME overlaps
! the position of each occurrence of the SCREEN_ORDER element
! in the infobuf. Thus setting each FIELD_NAME element to
! a literal that is 16 characters long or Less results in each
! SCREEN_ORDER element being implicitly filled with blanks.
!
                  FIELD_NAME(1,1)   = "BADGE_NUMBER"
!
                  FIELD_NAME(1,2)   = "LAST_NAME"
!
                  FIELD_NAME(1,3)   = "SUR_NAME"
!
!       Now determine the length of the entire Fieldinfo buffer.
!
                INFOBUFLEN = (NUM_ENTRIES * ENTRY_LEN) + 10
!
                CALL VGETFIELDINFO (COMAREA,
          +                         FIELDINFO,
          +                         INFOBUFLEN)
!
                IF (CSTATUS.NE.0) THEN
                  STOP_NOW = 1
                ERROR_LOCATION =
            +    "**** Routine: Setup For Work - Field Info Retrieval"
                CALL GET_ERROR_MESSAGE
             END IF
           END IF
!
           END
```

```
$TITLE '            Collect Transactions'
!***************************************************************!
!                                                               !
!                 Collect Transactions                          !
!                                                               !
!***************************************************************!

!
      SUBROUTINE COLLECT_TXNS
!
      IMPLICIT NONE
!
      COMMON /COMO1/   COMAREA
      COMMON /COMll/   STOP_NOW
      COMMON /COM12/   DONE_WITH_TXNS
      COMMON /COMI3/   NBR_TXN_COLLECTED
      COMMON /COM21/   FIELDINFO
      COMMON /COM81/   MSGBUF
      COMMON /COM82/   MSGBUFLEN
      COMMON /COM83/   ERRMSGLEN
      COMMON /COM91/   ERROR_LOCATION
      COMMON /COM101/  FOUND_DATA_ERRS
!
      SYSTEM INTRINSIC VCHANGEFIELD,
     +                   VPUTWINDOW,
     +                   VINITFORM,
     +                   VSHOWFORM
!
      INTEGER*2      COMAREA(60)
      INTEGER*2      CSTATUS
      INTEGER*2      REPEATAPP
      INTEGER*2      FREEZEAPP
      CHARACTER*16   NFNAME
      EQUIVALENCE   (COMAREA(1),  CSTATUS),
     +              (COMAREA(27), REPEATAPP),
     +              (COMAREA(28), FREEZEAPP),
     +              (COMAREA(19), NFNAME)
      INTEGER*2      STOP_NOW
      INTEGER*2      DONE_WITH_TXNS
      INTEGER*2      NBR_TXN_COLLECTED
      INTEGER*2      FIELDINFO(37)
      INTEGER*2      NUM_ENTRIES
      INTEGER*2      ENTRY_LEN
      CHARACTER*16   FORM_NAME
      EQUIVALENCE   (FIELDINFO(1),  NUM_ENTRIES),
     +              (FIELDINFO(2),  ENTRY_LEN),
     +              (FIELDINFO(3),  FORM_NAME)
             INTEGER*2   FIELD NAME (9,3)
             INTEGER*2   SCREEN ORDER (9,3)
             EQUIVALENCE  (FIELDINFO(11), FIELD_NAME),
     +                    (FIELDINFO(11), SCREEN_ORDER)
      CHARACTER*150 MSGBUF
      INTEGER*2      MSGBUFLEN
      INTEGER*2      ERRMSGLEN
      CHARACTER*70   ERROR_LOCATION

               INTEGER*2    FIELDSPECS(12)
                    INTEGER*2     FIELD_ID(4,3)
```

```
                  INTEGER*2     CHANGE_TYPE(4,3)
                  CHARACTER*4   CHANGE_SPEC(2,3)
                  EQUIVALENCE (FIELDSPECS (1),  FIELD_ID) ,
         +                    (FIELDSPECS (1),  CHANGE_TYPE),
         +                    (FIELDSPECS (1),  CHANGE_SPEC)
         INTEGER*2     NUMSPECS
         INTEGER*2     FOUND_DATA_ERRS
!
! Setup for and get transaction data entry form.
!
         REPEATAPP = 0
         FREEZEAPP = 0
!
         NFNAME = "DEDUCTION"
!
         CALL VGETNEXTFORM (COMAREA)
!
         IF (CSTATUS.NE.0) THEN
             STOP_ NOW = 1
             ERROR_LOCATION =
        +     "**** Routine: Collect Transactions - Form Retrieval"
             CALL GET_ERROR_MESSAGE
         END IF
!
! Toggle three fields in form to "input allowed".
!
!      Screen order is indicated to field change intrinsic
!      as a negative number.
!
!      Change field type is indicated by a 5.
!
!      "Input allowed" is indicated by an "O" (for input/output).
!
         IF (STOP_NOW.EQ.0) THEN
!
               FIELD_ID(1,1)     = (SCREEN_ORDER(9,1) * (-1))
               CHANGE_TYPE(2,1)  = 5
               CHANGE_SPEC (2,1) = "O"
!
               FIELD ID(1,2)     = (SCREEN_ORDER(9,2) * (-1))
               CHANGE_TYPE(2,2)  = 5
               CHANGE_SPEC(2,2)  = "O"
!
               FIELD_ID(1,3)     = (SCREEN_ORDER(9,3) * (-1))
               CHANGE_TYPE(2,3)  = 5
               CHANGE_SPEC(2,3)  = "O"
!
               NUMSPECS = 3
!
               CALL VCHANGEFIELD (COMAREA,
        +                         FIELDSPECS,
        +                         NUMSPECS)
!
       IF (CSTATUS.NE.0) THEN
          STOP_NOW = 1

          ERROR_LOCATION =
           "**** Routine: Collect Transactions - Field Type Updates"
```

```
            CALL GET_ERROR_MESSAGE
         END IF
      END IF
!
! Load window message.
!
      IF (STOP_NOW.EQ.0) THEN
!
          MSGBUFLEN = 79
!
          MSGBUF =
    +       "Fill in Deduction Transaction according to worksheet."
!
          CALL VPUTWINDOW (COMAREA,
    +                          MSGBUF,
    +                          MSGBUFLEN)
!
          IF (CSTATUS.NE.0) THEN
             STOP_NOW = 1
             ERROR_LOCATION =
    +         "**** Routine: Collect Transactions - Window Load"
             CALL GET_ERROR_MESSAGE
          END IF
      END IF
!
! Initialize form.
!
      IF (STOP_NOW.EQ.0) THEN
!
         CALL VINITFORM (COMAREA)
!
         IF (CSTATUS.NE.0) THEN
            STOP_NOW = 1
            ERROR_LOCATION =
      +      "**** Routine: Collect Transactions - Form Init"
            CALL GET_ERROR_MESSAGE
         END IF
      END IF
!
! Show form.
!
      IF (STOP_NOW.EQ.0) THEN
!
        CALL SHOWFORM (COMAREA)
!
        IF (CSTATUS.NE.0) THEN
           STOP_NOW = 1
           ERROR_LOCATION =
       +    "**** Routine: Collect Transactions - Form display"

            CALL GET_ERROR_MESSAGE
         END IF
      END IF
!
! Setup and loop on transaction until it can be filed.
!
      FOUND_DATA_ERRS = 1
!
```

```
      DO WHILE (FOUND_DATA_ERRS.EQ.1
      +    .AND.STOP_NOW.EQ.0
      +    .AND.DONE_WITH_TXNS.EQ.0)
!
         CALL READ_EDIT_AND_FILE
!
      END DO
!
      END
```

```
$TITLE '             Read Edit and File'

!***************************************************************!
!                                                               !
!                  Read Edit and File                           !
!                                                               !
!***************************************************************!
!
      SUBROUTINE READ_EDIT_AND_FILE
!
      IMPLICIT NONE
!
      COMMON /COM01/   COMAREA
      COMMON /COMll/   STOP_NOW
      COMMON /COM12/   DONE_WITH_TXNS
      COMMON /COM13/   NBR_TXN_COLLECTED
      COMMON /COM81/   MSGBUF
      COMMON /COM82/   MSGBUFLEN
      COMMON /COM83/   ERRMSGLEN
      COMMON /COM91/   ERROR_LOCATION
      COMMON /COM101/  FOUND_DATA_ERRS
!
      SYSTEM INTRINSIC VREADFIELDS,
     +                 VFIELDEDITS,
     +                 VFINISHFORM,
     +                 VGETBUFFER
!
      INTEGER*2       COMAREA(60)
      INTEGER*2       CSTATUS
      INTEGER*2       LASTKEY
      INTEGER*2       NUMERRS
      EQUIVALENCE   (COMAREA(1),  CSTATUS),
     +              (COMAREA(6),  LASTKEY),
     +              (COMAREA(7),  NUMERRS)
      INTEGER*2       STOP_NOW
      INTEGER*2       DONE_WITH_TXNS
      INTEGER*2       NBR_TXN_COLLECTED
      CHARACTER*150 MSGBUF
      INTEGER*2       MSGBUFLEN
      INTEGER*2       ERRMSGLEN
      CHARACTER*70    ERROR_ LOCATION
      INTEGER*2       FOUND_ DATA_ERRS
      CHARACTER*200   DATABUF
      INTEGER*2       DATABUFLEN
!
! Read form.
!
      CALL VREADFIELDS (COMAREA)
!
      IF (CSTATUS.NE.0) THEN
         STOP_NOW = 1
         ERROR_LOCATION =
     +    "**** Routine: Read Edit and File - Terminal Read"
         CALL GET_ERROR_MESSAGE
      END IF
!
! Determine if operator wants to stop transaction collection.
!
```

```
      IF (STOP NOW.EQ.0) THEN
          IF (LASTKEY.EQ.8) THEN
             DONE_WITH_TXNS = 1
          END IF
      END IF
!
! Edit data read from terminal.
!
      IF        (STOP_NOW.EQ.0
      +   .AND.DONE_WITH_TXNS.EQ.0) THEN
!
          CALL VFIELDED (COMAREA)
!
          IF (CSTATUS.NE.0) THEN
               STOP_NOW = 1
               ERROR_LOCATION =
      +         "**** Routine: Read Edit and File - Data Edit"
               CALL GET_ERROR_MESSAGE
          END IF
      END IF
!
! Determine if edit errors detected.
!
      IF        (STOP_NOW.EQ.0
      +   .AND.DONE_WITH_TXNS.EQ.0) THEN
!
          IF (NUMERRS.LT.1) THEN
               FOUND_DATA_ERRS = 0
          END IF
      END IF
!
! Finish form data.
!
      IF        (STOP_NOW.EQ.0
      +   .AND.DONE_WITH_TXNS.EQ.0
      +   .AND.FOUND_DATA_ERRS.EQ.0) THEN
!
           CALL VFINISHFORM (COMAREA)
!
           IF (CSTATUS.NE.0) THEN
               STOP NOW = 1
               ERROR_LOCATION =
      +         "**** Routine: Read Edit and File - Data Finishing"
               CALL GET_ERROR_MESSAGE
          END IF
      END IF
!
! Determine if data finishing errors detected.
!
      IF       (STOP_NOW.EQ.0
      +   .AND.DONE_WITH_TXNS.EQ.0
      +   .AND.FOUND_DATA_ERRS.EQ.0) THEN
!
          IF (NUMERRS.GT.0) THEN
             FOUND_DATA_ERRS = 1
          END IF
       END IF
```

```
!
! Do we have a transaction that can be filed?
!
       IF    (STOP_NOW.EQ.0
      +   .AND.DONE_WITH_TXNS.EQ.0) THEN
!
          IF   (FOUND_DATA_ERRS.EQ.0
      +      .AND.LASTKEY.EQ.0) THEN
!
! Get transaction from form and file it.
!
             DATABUF = " "
!
             DATABUFLEN = 200
!
             CALLVGETBUFFER (COMAREA,
      +                      DATABUF,
      +                      DATABUFLEN)
!
             IF (CSTATUS.NE.0) THEN
                STOP NOW = 1
                ERROR_LOCATION =
      +          "**** Routine: Read Edit and File - Data Get"
                CALL GET_ERROR_MESSAGE
!
             ELSE
!
!    Write Databuf to Transaction File.
!
                WRITE (UNIT = 10,
      +                ERR  = 310) DATABUF
!
                GOTO 320
!
310             STOP_NOW = 1
                ERROR_LOCATION =
      +          "**** Routine: Read Edit and File - File Write"
                MSGBUF =
      +          "**** Write to Transaction File failed!"
!
320            IF (STOP_NOW.EQ.0) THEN
                  NBR_TXN_COLLECTED = NBR_TXN_COLLECTED + 1
               END IF
            END IF
         END IF
      END IF
!
! Do we need to prompt the operator to correct errors?
!
      IF    (STOP_NOW.EQ.O
    +    .AND.DONE_WITH_TXNS.EQ.0) THEN
!
         IF    (FOUND DATA ERRS.EQ.1
    +      .AND.LASTKEY.EQ._O) THEN
!
            CALL PROMPT _OPERATOR
!
         END IF
      ENDIF
!
! Do we need to refresh the display?
!
      IF    (STOP_ NOW.EQ.0
    +    .AND.DONE_WITH_TXNS.EQ.0) THEN
!
         IF    (FOUND_DATA_ERRS.EQ.1
    +      .AND.LASTKEY.NE.0) THEN
!
!  The operator pressed some key other than <ENTER>
!  or <EXIT> so clear data error flag to break loop.
!
            FOUND_ DATA_ERRS = 0
!
         END IF
      END IF
!
      END
```

```
$TITLE '                 Prompt Operator'

!***************************************************************!
!                                                               !
!                        Prompt Operator                        !
!                                                               !
!***************************************************************!
!
      SUBROUTINE PROMPT_OPERATOR
!
      IMPLICIT NONE
!
      COMMON /COMO1/ COMAREA
      COMMON /COM11/ STOP NOW
      COMMON /COM81/ MSGBUF
      COMMON /COM82/ MSGBUFLEN
      COMMON /COM83/ ERRMSGLEN
      COMMON /COM91/ ERROR_LOCATION
!
      SYSTEM INTRINSIC VPUTWINDOW,
     +                 VSHOWFORM
!
      INTEGER*2       COMAREA(60)
      INTEGER*2       CSTATUS
      EQUIVALENCE (COMAREA(1), CSTATUS)
      INTEGER*2       STOP_NOW
      CHARACTER*150   MSGBUF
      INTEGER*2       MSGBUFLEN
      INTEGER*2       ERRMSGLEN
      CHARACTER*70    ERROR_LOCATION
!
! Get message text associated with first field flagged
! with a data error.
!
       CALL GET_ERROR_MESSAGE

       CALL VPUTWTNDOW (COMAREA,
     +                  MSGBUF,
     +                  ERRMSGLEN)
!
       IF (CSTATUS.NE.0) THEN
          STOP NOW = 1
          ERROR_LOCATION =
     +     "**** Routine: Prompt Operator - Window Load"
          CALL GET_ERROR_MESSAGE
       END IF
!
! Display highlighted form and updated window message.
!
       IF (STOP_NOW.EQ.0) THEN
          CALL VSHOWFORM (COMAREA)
!
       IF (CSTATUS.NE.0) THEN

            STOP_NOW = 1
            ERROR_LOCATION =
     +       "**** Routine: Prompt Operator - Display Updates"
            CALL GET_ERROR_MESSAGE
            END IF
        END IF
!
        END
```

```
$TITLE '                Cleanup After Work'

!***************************************************************!
!                                                               !
!                       Cleanup After Work                      !
!                                                               !
!***************************************************************!
!
           SUBROUTINE CLEANUP_AFTER_WORK
!
           IMPLICIT NONE
!
           COMMON /COMO1/ COMAREA
!
           SYSTEM INTRINSIC VCLOSEFORMF,
          +                 VCLOSETERM
!
           INTEGER*2     COMAREA(60)
           INTEGER*2     CSTATUS
           EQUIVALENCE (COMAREA(1), CSTATUS)
!
! Note that this routine unconditionally attempts to close
! the Forms File and Terminal
!
           CLOSE (UNIT = 10)
!
           CSTATUS = 0
!
           CALL VCLOSEFORMF (COMAREA)
!
           CSTATUS = 0
!
           CALL VCLOETERM (COMAREA)
!
           END
```

```
$TITLE '                 Get Error Message'

!***************************************************************!
!                                                               !
!                        Get Error Message                      !
!                                                               !
!***************************************************************!
!
         SUBROUTINE GET-ERROR-MESSAGE
!
         IMPLICIT NONE
!
         COMMON /COM01/ COMAREA
         COMMON /COM81/ MSGBUF
         COMMON /COM82/ MSGBUFLEN
         COMMON /COM83/ ERRMSGLEN
!
         SYSTEM INTRINSIC VERRMSG
!
         INTEGER*2 COMAREA(60)
         CHARACTER*150 MSGBUF
         INTEGER*2 MSGBUFLEN
         INTEGER*2 ERRMSGLEN
!
         MSGBUF = " "
         MSGBUFLEN = 150
!
         CALL VERRMSG (COMAREA,
        +              MSGBUF,
        +              MSGBUFLEN,
        +              ERRMSGLEN)
!
         END
```

```
$TITLE '              Display System Error'

!***************************************************************!
!                                                               !
!                    Display System Error                       !
!                                                               !
!***************************************************************!
!
     SUBROUTINE DISPLAY_SYSTEM_ERROR
!
     IMPLICIT NONE
!
     COMMON /COM81/ MSGBUF
     COMMON /COM91/ ERROR_LOCATION
!
     CHARACTER*150 MSGBUF
     CHARACTER*70 ERROR_LOCATION
!
     PRINT *,
    + "**** Transaction entry facility detected system error at:"
     PRINT *, ERROR_LOCATION
     PRINT *,
    + "**** The error message returned is:"
     PRINT *, MSGBUF
!
     END
```

BASIC

```
 60 REM   This program controls source data entry for any forms file.
 65 REM   It prompts the user for the name of a forms file and opens it
 70 REM   if possible. It also prompts for the name of a BATCH file and
 75 REM   opens it if possible. If all is OK, it displays the HEAD FORM
 80 REM   accepts input, edits the data, and if no errors, writes data
 85 REM   to the BATCH file. This process continues until the form with
 90 REM   name $END is encountered or the EXIT function key is pressed.
 95 REM
100 REM   This program also provides BROWSE control and permits
105 REM   modification of the collected data.
110 REM
115 REM   The function keys have the following meanings:
120 REM
125 REM           f1       f2          f3        f4
130 REM          HEAD     DELETE       PRINT    REFRESH
135 REM
140 REM           f5        f6         f7         f8
145 REM          PREV      NEXT      BROWSE/     EXIT
150 REM                             COLLECT
155 REM
160 REM   The following variable assignments are used:
165 REM
170 REM   Strings:
175 REM
180 REM     B$[16]     - Name of current Form
185 REM     BO$[16]    - Saves B$ during BROWSE
190 REM     B1$[16]    - Name of next Form
195 REM     E$[60]     - Entry error messages
200 REM     M$[150]    - General messages
205 REM     S$[150]    - Status line
210 REM     U$[150]    - General user input and file names
220 REM
225 REM  Integers:
230 REM
235 REM     B1   - Batch : O=false; 1=true
240 REM     F1   - First time : O=false; 1=true
245 REM     W1   - Max window length (150)
250 REM     R1   - Saves number of last accessed record in batch file
255 REM     D1   - Browse direction: O=forward; 1=backward
260 REM     R2 -   Local batch record number
265 REM
270 REM     E    - Errors Flag : O=false; 1=true
275 REM
280 REM
285 REM  Com area Array C(1:60)
295 REM     C(1)    - Status (O=OK; >0 ERROR)
300 REM     C(2)    - Language (1=BASIC)
305 REM     C(3)    - Com area length (60)
310 REM     C(4)    - Com extention length (2000)
315 REM     C(5)    - Mode
320 REM     C(6)    - Lastkey (# of last used function key)
325 REM     C(7)    - Numerrs
330 REM     C(8:10)    - not used
335 REM    C(11:18) - name of current form (packed)
```

```
 340 REM    C(19:26) - name of next form (packed)
 345 REM    C(27) - Repeat option
 350 REM    C(28) - NF option
 355 REM    C(29) - not used
 360 REM    C(30) - Length of data buffer
 365 REM    C(31) - not used
 370 REM    C(33) - Delete flag
 375 REM    C(34) - Show control
 80 REM    C(35:42) - not used
 385 REM    C(43:44) - Number of recs in batch file (double)*
 390 REM    C(45:46) - Record # in batch file (double)*
 395 REM    C(47,48) - not used
 400 REM    C(49) - Terminal file #
 405 REM    C(56) - Terminal options
 410 REM
 415 REM      *Only c(44) and c(46) are used in this program.
 420 REM       Therefore, this program cannot handle BATCH files
 425 REM       that have more than 32768 records.
 430 REM
 450 REM
 451 REM *************************************************************
 452 REM
 500 REM2 ** DECLARATIONS **
 510 DIM B$[16],BO$[16],B1$[16],C$[2]
 520 DIM E$[60],M$[150],S$[150],U$[150],X$[2]
 530 INTEGER B1,D1,F1,E,I,J,R1,R2,W1,X,Y,P,P1
 540 INTEGER C[4560]
 900 REM2 ** FUNCTIONS **
 905 REM1 * UNPACK INTEGER INTO 2 CHARACTERS *
 910 DEF FNU$(X)
 915   Y=INT(X/256)
 920   X=X-256*Y
 925   RETURN CHR$(Y)+CHR$(X)
 930 FNEND
 950 REM1 * PACK 2 CHARACTERS INTO INTEGER *
 955 DEF INTEGER FNP(X$)
 960   RETURN NUM(X$[2;1])+NUM(X$[1;1])*256
 965 FNEND
1000 REM2 ** ENTRY **
1010 REM *INITIALIZE*
1020 GOSUB 2000
1030 REM1 *COLLECT*
1040 GOSUB 3000
1050 REM1 *EXIT*
1060 GOSUB 5000
1070 END
2000 REM2 <INITIALIZE>
2010 MAT C=ZER
2020 C[2]=l
2030 C[3]=60
2040 C[4]=2000
2050 B1=1
2060 E=O
2070 W1=150
2100 REM1 *OPEN FORMS FILE*
2110 C[1]=0
2120 PRINT " Enter FORMS file name and press RETURN: ";
2130 LINPUT U$
```

```
2140 IF NOT LEN(U$) THEN 9900
2150 U$=U$+" "
2160 CALL VOPENFORMF(C[*],U$)
2170 IF C[1] THEN DO
2180   GOSUB 9000
2190   GOTO 2100
2200 DOEND
2210 REM1 *OPEN BATCH FILE*
2220 PRINT " Enter BATCH file name and press RETURN: ";
2230 LINPUT U$
2240 IF NOT LEN(U$) THEN B1=0
2250 ELSE DO
2260  U$=U$+" "
2270  CALL VOPENBATCH(C[*],U$)
2280  IF C[1] THEN DO
2290  IF C[1]=70 OR C[1]=73 THEN DO
2300   IF C[1]=70 THEN PRINT "WARNING: A different FORMS file was used to
2305              create this batch."
2310    IF C[1]=73 THEN PRINT "WARNING: FORMS file was recompiled since
2315               this batch was created.
2320    PRINT "Enter "'34"Y"'34" to continue: ";
2330    LINPUT U$
2340    IF UPS$(U$)="Y" THEN C[1]=0
2350     DOEND
2360     ELSE GOSUB 9000
2370   DOEND
2380 DOEND
2390 REM2 ** OPEN TERMINAL **
2400 IF NOT B1 OR NOT C[1] THEN DO
2410    U$="A264X"
2420    CALL VOPENTERM(C[*],U$)
2430    IF C[1] THEN DO
2440      GOSUB 9000
2450      END
2460    DOEND
2470    C[56]=C[56]+8
2480    RETURN
2490 DOEND
2500 ELSE DO
2510    C[1]=0
2520    CALL VCLOSEBATCH(C[*])
2530    CALL VCLOSEFORMF(C[*])
2540 DOEND
2550 GOTO 2100
3000 REM2 <COLLECT>
3005 F1=1
3010 C[5]=C[33]=0
3015 IF NOT C[6] OR C[6]=6 THEN DO
3020    IF NOT C[27] AND NOT C[28] OR C[27]=2 THEN DO
3025     CALL VSHOWFORM(C[*])
3030     IF C[1] THEN GOSUB 9100
3035   DOEND
3040 DOEND
3045 REM1 *COLLECT LOOP*
3050 CALL VGETNEXTFORM(C[*])
3055 IF F1 AND C[1] THEN DO
3060    CALL VERRMSG(C[*],M$,BO,I
3065    E=1
```

```
3070    RETURN
3075 DOEND
3080 IF C[1] THEN GOSUB 9100
3085 F1=0
3090 CALL VINITFORM(C[*])
3095 IF C[1] OR C[7] THEN GOSUB 9100
3100 IF NOT E THEN GOSUB 8000
3105 REM1 *SOFTKEY LOOP*
3110 E=O
3115 CALL VSHOWFORM(C[*])
3120 IF C[1] THEN GOSUB 9100
3125 C[34]=0
3130 IF C[30]<=0 AND C[27]=0 AND C[28)<>0 THEN DO
3132   IF NOT E AND B1 THEN DO
3135     CALL VWRITEBATCH(C[*])
3140     IF C[1] THEN GOSUB 9100
3145     IF NOT E THEN DO
3146       C[46]=C[46]+1
3147       P1=20
3148       P=C[46] MOD P1
3149       IF P=0 THEN CALL VPOSTBATCH(C[*])
3150     DOEND
3151   DOEND
3152 DOEND
3155 ELSE DO
3160   CALL VREADFIELDS(C[*])
3165   IF C[1] THEN GOSUB 9100
3170   IF NOT E AND C[6]=8 THEN RETURN
3175   IF NOT E THEN GOSUB C[6]+1 OF 3300,3350,3400,3450,3500,3550,3600,370
3180 DOEND
3185 IF C[6]=8 THEN RETURN
3190 IF E OR C[6]=3 THEN 3105
3195 GOSUB 8100
3200 IF B1$<>"$END" OR C[27] THEN 3045
3205 RETURN
3300 REM2 <ENTER KEY>
3305 CALL VFIELDEDITS(C[*])
3310 IF C[1] OR C[7] THEN GOSUB 9100
3315 IF NOT E THEN DO
3320   CALL VFINTSHFORM(C[*])
3325   IF C[1] THEN GOSUB 9100
3330   IF NOT E AND B1 THEN DO
3335     CALL VWRITEBATCH(C[*])
3340     IF C[1] THEN GOSUB 9100
3342     IF NOT E THEN DO
3343        C [46]=C[46]+1
3344        P1 = 20
3345        P=C[46] MOD P1
3346        IF P=O THEN CALL VPOSTBATCH(C[*])
3347      DOEND
3348   DOEND
3349 DOEND
3350 RETURN
3351 REM1 <HEAD KEY>
3355 C[27]=C[28]=O
3360 B1$="$HEAD"
3365 GOSUB 8200
3370 RETURN
```

```
3400 REM1 <DELETE KEY>
3410 E$="DELETE key defined only for BROWSE"
3420 GOSUB 9200
3430 RETURN
3450 REM1 <PRINT KEY>
3455 I=1
3460 J=49
3465 CALL VPRINTFORM(C[*],I,J)
3470 IF C[1] THEN GOSUB 9100
3475 RETURN
3500 REM1 <REFRESH KEY>
3505 B1$="$REFRESH"
3510 GOSUB 8200
3515 RETURN
3550 REM1 <PREV KEY>
3555 E$=" The PREV key is only defined for BROWSE mode."
3560 GOSUB 9200
3565 RETURN
3600 REM1 <NEXT KEY>
3610 IF NOT C[27] THEN DO
3620    E$=" The NEXT key is not defined for a non-repeating form."
3630    GOSUB 9200
3640 DOEND
3650 ELSE C[27]=0
3660 RETURN
3700 REM1 <BROWSE KEY>
3710 IF NOT B1 THEN DO
3720    E$=" No BATCH file was specified, so BROWSE is not allowed."
3730    GOSUB 9200
3740 DOEND
3750 ELSE DO
3760    IF NOT C[44] THEN DO
3770       E$=" There are no more batch records."
3780       GOSUB 9200
3790    DOEND
3800    ELSE DO
3810       R1=C[46]
3820       GOSUB 8100
3830       BO$=B$
3840       C[5]=1
3850       C[27]=C[28]=0
3860     GOSUB 4000
3870     C[5]=0
3880     B1$=B0$
3890     GOSUB 8200
3900     C[46]=R1
3910     C[27]=C[28]=C[33]=0
3920     IF C[6]=8 THEN DO
3921       B$=B0$
3922       RETURN
3923     DOEND
3930   DOEND
3940 DOEND
3950 RETURN
4000 REM2 <BROWSE>
4005 R2=C[46]
4010 C[46]=C[46]-1
4015 D1=1
```

```
4020 REM1 *BROWSE UNTIL EXIT OR COLLECT KEY*
4025 IF NOT C[44] THEN RETURN
4030 IF C[46]=R1 THEN DO
4035   E$=" There are no more batch records."
4040   GOSUB 9200
4045   C[46]=C[46]-1
4050   D1=1
4055 DOEND
4060 IF C[46]<0 THEN DO
4065   E$=" There are no previous batch records."
4070   GOSUB 9200
4075   D1=C[46]=0
4080 DOEND
4085 CALL VREADBATCH(C[*])
4090 IF C[1] THEN GOSUB 9100
4095 IF NOT C[33] THEN DO
4100   IF C[46]<>R2 OR C[6]=4 THEN DO
4105     IF D1 OR C[6]=4 THEN C[27]=C[28]=0
4110     ELSE DO
4115       GOSUB 8100
4120       IF B$<>B1$ THEN C[27]=0
4125     DOEND
4130     IF C[6]=4 THEN DO
4135       B1 $="$REFRESH"
4140       GOSUB 8200
4145     DOEND
4150     CALL VGETNEXTFORM(C[*])
4155     IF C[1] THEN GOSUB 9100
4160     R2=C[46]
4165   DOEND
4170   IF NOT E THEN GOSUB 8000
4175   REM2 *SOFTKEY LOOP*
4180   E=0
4185   CALL VSHOWFORM(C[*])
4190   IF C[1] THEN GOSUB 9100
4195   C[34]=0
4200   CALL VREADFIELDS(C[*])
4205   IF C[1] THEN GOSUB 9100
4210   IF NOT E AND C [6] >6 THEN RETURN
4215   IF NOT E THEN GOSUB C[6]+1 OF 4300,4400,4450,4500,4550,4600,4650
4220   IF E OR C[6]=3 THEN 4175
4225 DOEND
4230 ELSE DO
4235   IF D1 THEN C[46]=C[46]-1
4240   ELSE C[46]=C[461+1
4245 DOEND
4250 GOTO 4020
4255 RETURN
4300 REM2 <ENTER KEY>
4305 D1=0
4310 CALL VFIELDEDITS(C[*])
4315 IF C[1] 0 C[7] THEN GOSUB 9100
4320 IF NOT E THEN DO
4325   CALL VFINISHFORM(C[*])
4330   IF C[1] THEN GOSUB 9100
4335   IF NOT C[27] AND C[28] OR C[27]=2 THEN DO
4340     CALL VSHOWFORM(C[*])
4345     IF C[1]THEN GOSUB 9100
```

```
4350   DOEND
4355   IF NOT E THEN DO
4360     CALL VWRITEBATCH(C[*])
4365     IF C[1] THEN GOSUB 9100
4370     IF NOT E THEN C[46]=C[46]+1
4375   DOEND
4380 DOEND
4385 RETURN
4400 REM1 <HEAD KEY>
4410 D1=C[27]=C[28]=C[46]=0
4420 RETURN
4450 REM1 <DELETE KEY>
4455 D1=0
4460 C[33]=1
4465 CALL VWRITEBATCH(C[*])
4470 IF C[1] THEN GOSUB 9100
4475 C[33]=0
4480 IF NOT E THEN C[46]=C[46]+1
4485 C[27]=C[28]=0
4490 RETURN
4500 REM1 <PRINT KEY>
4505 I=1
4510 J=49
4515 CALL VPRINTFORM(C[*],I,J)
4520 IF C[1] THEN GOSUB 9100
4525 RETURN
4550 REM1 <REFRESH KEY>
4560 RETURN
4600 REM1 <PREV KEY>
4610 D1=1
4620 C[46]=C[46]-1
4630 RETURN
4650 REM1 <NEXT KEY>
4655 D1=0
4660 C[46]=C[46]+1
4665 IF NOT C[27] AND C[28] OR C[27]=2 THEN DO
4670   CALL VSHOWFORM(C[*])
4675   IF C[1] THEN GOSUB 9100
4680 DOEND
4685 RETURN
5000 REM2 <EXIT>
5010 REM1 *CLOSE TERMINAL*
5020 CALL VCLOSETERM(C[*])
5030 IF C[1] THEN GOSUB 9000
5040 C[1]=0
5050 REM1 *CLOSE BATCH FILE*
5060 IF B1 THEN DO
5070   IF E THEN PRINT M1$
5080   ELSE DO
5090     CALL VCLOSEBATCH(C[*])
5100     IF C[1] THEN GOSUB 9100
5110     C[1]=0
5120   DOEND
5130 DOEND
5140 REM1 *CLOSE FORMS FILE*
5150 CALL VCLOSEFORMF(C[*])
5160 IF C[1] THEN GOSUB 9100
5170 C[1]=0
```

```
5180 RETURN
8000 REM2 <PRINT STATUS LINE>
8010 S$=" ENTRY "+V$+'27"&a31CBATCH RECORD # "
8020 CONVERT C[46]+1 TO S$[LEN(S$)+1]
8030 S$=S$+'27"&a65CMODE: "
8040 IF NOT C[5] THEN S$=S$+"COLLECT"
8050 ELSE S$=S$+'27"&dKBROWSE"
8060 I=LEN(S$)
8070 CALL VPUTWINDOW(C[*],S$,I)
8080 RETURN
8100 REM2 <GET FORM NAMES FROM COM ARRAY>
8110 B$=B1$=""
8120 FOR K=11 TO 18
8130   B$=B$+FNU$(C[K])
8140   B1$=B1$+FNU$(C[K+8])
8150 NEXT K
8160 B$=DEB$(B$[1,15])
8170 B1$=DEB$(B1$[1,15])
8180 RETURN
8200 REM2 <PUT NEXT FORM NAME IN COM ARRAY>
8210 B1$(LEN(B1$)+1,16] =""
8220 B1$[16,16]='0
8230 FOR K=1 TO 15 STEP 2
8240   C[INT(K/2)+19]=FNP(B1$[K;2])
8250 NEXT K
8260 RETURN
9000 REM2 <PRINT ERROR MESSAGE TO SCREEN>
9020 CALL VERRMSG(C[*],M$,W1,I)
9030 PRINT M$
9040 RETURN
9100 REM2 <PRINT VIEW ERROR MESSAGE TO DISPLAY WINDOW>
9105 IF E THEN RETURN
9110 E=1
9125 CALL VERRMSG(C[*],M$,W1,I)
9130 M$=" "+M$
9135 I=LEN(M$)
9140 C[1]=0
9145 CALL VPUTWINDOW(C[*],M$,I)
9150 RETURN
9200 REM2 <PRINT ENTRY ERROR MESSAGE TO DISPLAY WINDOW>
9210 IF E THEN RETURN
9220 E=1
9230 I=LEN(E$)
9240 CALL VPUTWINDOW(C[*],E$,I)
9250 RETURN
9900 REM2 ** TERMINATE **
9910 PRINT "END OF PROGRAM"
9920 END
```

TRANSACT

```
SYSTEM TRANS,ENTITY =TXNFILE(APPEND);
<<
This application collects employee payroll deduction transactions
and places the edited transactions into a file.

For this application:        Enter key = edit and file transaction;

                                    f8 = exit application;

                      all other f keys = redo transaction.

Each transaction entered by the operator is subjected to the data
edits embedded within the input form.

The application continues to collect transactions until either
the operator signals to exit or a system error is detected.
>>
DEFINE(ITEM) MAXWINDOWLEN I(4),INIT=150;
DEFINE(ITEM) COMAREA 60 I(4):
                CSTATUS I(4)=COMAREA:
                LANGUAGE I(4)=COMAREA(3):
                COMAREALEN I(4)=COMAREA(5):
                USERBUFLEN I(4)=COMAREA(7):
                CMODE I(4)=COMAREA(9):
                LASTKEY I(4)=COMAREA(11):
                NUMERRS I(4)=COMAREA(13):
                WINDOWENH I(4)=COMAREA(15):
                MULTIUSAGE I(4)=COMAREA(17):
                LABELOPTIONS I(4)=COMAREA(19):
                CFNAME X(16)=COMAREA(21):
                NFNAME X(16)=COMAREA(37):
                REPEATAPP I(4)=COMAREA(53):
                FREEZEAPP I(4)=COMAREA(55):
                CFNUMLINES I(4)=COMAREA(57):
                DBUFLEN I(4)=COMAREA(59):
                LOOKAHEAD I(4)=COMAREA(63):
                DELETEFLAG I(4)=COMAREA(65):
                SHOWCONTROL I(4)=COMAREA(67):
                PRINTFILENUM I(4)=COMAREA(71):
                FILERRNUM I(4)=COMAREA(73):
                ERRFILENUM I(4)=COMAREA(75):
                FORMSTORESIZE I(4)=COMAREA(77):
                NUMRECS I(8)=COMAREA(85):
                RECNUM I(8)=COMAREA(89):
                TERM-FILEN I(4)=COMAREA(97):
                RETRIES I(4)=COMAREA(109):
                TERM-OPTIONS I(4)=COMAREA(111):
                ENVIRON I(4)=COMAREA(113):
                USERTIME I(4)=COMAREA(115):
                IDENTIFIER I(4)=COMAREA(117):
                LABELINFO X(2)=COMAREA(119):
              DATABUF X(200):
```

```
              DATABUFLEN I(4):

              DONE-WITH-TRANS I(4):
              ERROR-LOCATION X(70):
              FIELDINFO X(80):
                NUM-ENTRIES I(4)=FIELDINFO:
                ENTRY-LEN I(4)=FIELDINFO(3):
                FORM-NAME X(16)=FIELDINFO(5):
                FIELDDATA 3 X(20)=FIELDINFO(21):
                FIELDENTRY X(20)=FIELDDATA:
                  FIELD-NAME X(16)=FIELDENTRY:
                  SCREEN-ORDER I(4)=FIELDENTRY(17):
                  FIELD-NUM I(4)=FIELDENTRY(19):
              FIELDSPECS-ITEM 3 X(8):
                FIELDSPECS X(8)=FIELDSPECS-ITEM:
                  FIELD-ID I(4)=FIELDSPECS:
                  CHANGE-TYPE I(4)=FIELDSPECS(3):
                  CHANGE-SPEC X(4)=FIELDSPECS(5):
              FILENAME X(86):
              FOUND-DATA-ERRS I(4):
              INFOBUFLEN I(4):
              MSGBUF X(150):
              MSGBUFLEN I(4):
              NBR-TXN-COLLECT I(8):
              NUMENTRIES I(4):
              NUMSPECS I(4):
              STOP-NOW I(4):
              GLOBAL 3 I(4):
                ZERO I(4)=GLOBAL:
                ONE I(4)=GLOBAL(3):
                EIGHT I(4)=GLOBAL(5):
                FALSE I(4)=ZERO:
                TRUE I(4)=ONE;
LIST MAXWINDOWLEN;
LIST COMAREA,INIT:       << Initialize to all zeros >>
      DATABUF:
      DATABUFLEN:
      DONE-WITH-TRANS:
      ERROR-LOCATION:
      FIELDINFO:
      FIELDSPECS-ITEM:
      FILENAME:
      FOUND-DATA-ERRS:
      INFOBUFLEN:
      MSGBUF:
      MSGBUFLEN:
      NBR-TXN-COLLECT:
      NUMENTRIES:
      NUMSPECS:
      STOP-NOW:
      GLOBAL;
SET(OPTION) NOHEAD;

<< Sample program main line >>


PERFORM SETUP-FOR-WORK;
LEVEL;    <<setup loop until done collecting transactions>>
```

```
  IF (STOP-NOW) = (TRUE) THEN
    END(LEVEL)
  ELSE IF (DONE-WITH-TRANS) = (TRUE) THEN
    END(LEVEL)
  ELSE
    DO
    PERFORM COLLECT-TRANSACTIONS;
    END;                <<loop to next transaction>>
    DOEND;
PERFORM CLEANUP-AFTER-WORK;
DISPLAY "Deduction transactions collected this sesion =":
        NBR-TXN-COLLECT;
IF (STOP-NOW) = (TRUE) THEN
  PERFORM DISPLAY-SYSTEM-ERROR;
EXIT;

DISPLAY-SYSTEM-ERROR:
<<*****************>>

  DISPLAY "**** Transaction collection facility detected system "
          "error at: ":
          ERROR-LOCATION:
          "**** The error message returned is:",line=1:
          "****",line=1:
          MSGBUF;
  RETURN;

CLEANUP-AFTER-WORK:
<<***************>>

  FILE(CLOSE) TXNFILE;
  LET (CSTATUS) = (ZERO);
  PROC VCLOSEFORMF((COMAREA));
  LET (CSTATUS) = (ZERO);
  PROC VCLOSETERM((COMAREA));
  RETURN;

GET-ERROR-MESSAGE:
<<**************>>

  LET (MSGBUFLEN) = (MAXWINDOWLEN);
  MOVE (MSGBUF) = " ";
  PROC VERRMNSG((COMAREA),%(MSGBUF),(MSGBUFLEN),(MSGBUFLEN));
  RETURN;

PROMPT-OPERATOR:
<<************>>

  PERFORM GET-ERROR-MESSAGE;
  PROC VPUTWINDOW((COMPAREA),%(MSGBUF),(MSGBUFLEN));
  IF (CSTATUS) <> (ZERO) THEN
    DO

        LET (STOP-NOW) = (TRUE);
        MOVE (ERROR-LOCATION) = "**** procedure: Prompt Operator - "
                                "Window Load";
        PERFORM GET-ERROR-MESSAGE;
        RETURN;
        DOEND;
  PROC VSHOWFORM((COMAREA));
  IF (CSTATUS) <> (ZERO) THEN
        DO
        LET (STOP-NOW) = (TRUE);
        MOVE (ERROR-LOCATION) = "**** procedure: Collect Transactions"
                                " - Display Update";
        PERFORM GET-ERROR-MESSAGE;
        DOEND;
  RETURN;

READ-EDIT-FILE-TRANSACTION:
<<**********************>>

  PROC VREADFIELDS((COMAREA));
  IF (CSTATUS) <> (ZERO) THEN
        DO
        LET (STOP-NOW) = (TRUE);
        MOVE (ERROR-LOCATION) = "**** procedure: Read, Edit, and File"
                                " - Terminal Read";
        PERFORM GET-ERROR-MESSAGE;
        RETURN;
        DOEND;

  << Determine if operator wants to stop transaction collection >>

  IF (LASTKEY) = (EIGHT) THEN
        DO
        LET (DONE-WITH-TRANS) = (TRUE);
        RETURN;
        DOEND;

   IF (LASTKEY) <> (ZERO) THEN

        << Operator pressed some key other than Enter or Exit
           so clear data error flag to break loop           >>

        DO
        LET (FOUND-DATA-ERRS) = (FALSE);
        RETURN;
        DOEND;


    <<    Edit data   >>

    PROC VFIELDEDITS((COMAREA));
    IF (CSTATUS) <> (ZERO) THEN
      DO
      LET (STOP-NOW) = (TRUE);
      MOVE (ERROR-LOCATION) = "**** procedure: Read, Edit, and"
" File - data Edit";
         PERFORM GET-ERROR-MESSAGE;
         RETURN;
         DOEND;

     << Determine if edit errors >>

     IF (NUMERRS) < (ONE) THEN
```

```
          LET (FOUND-DATA-ERRS) = (FALSE);
      IF (FOUND-DATA-ERRS) = (FALSE) THEN
          DO

          << Finish form data >>

          PROC VFINISHFORM((COMAREA));
          IF (CSTATUS) <> (ZERO) THEN
             DO
             LET (STOP-NOW) = (TRUE);
             MOVE (ERROR-LOCATION) = "**** procedure: Read, Edit,"
                                     " and File - Data Finishing";
             PERFORM GET-ERROR-MESSAGE;
             RETURN;
             DOEND;
          IF (NUMERRS) > (ZERO) THEN
             LET (FOUND-DATA-ERRS) = (TRUE);
          DOEND; <<  Finish form data  >>

      << Do we have a transaction that can be filed? >>

      IF (FOUND-DATA-ERRS) = (FALSE) THEN
          DO

          << get transaction from form and file it >>

          MOVE (DATABUF) = " ";
          LET (DATABUFLEN) = 200;
          PROC VGETBUFFER((COMAREA),%(DATABUF),(DATABUFLEN));
          IF (CSTATUS) <> (ZERO) THEN
             DO
             LET (STOP-NOW) = (TRUE);
             MOVE (ERROR-LOCATION) = "**** procedure: Read, Edit, and"
                                     "File - Data Get";
             PERFORM GET-ERROR-MESSAGE;
             RETURN;
             DOEND;
          PUT TXNFILE,LIST=(DATABUF);
          LET (NBR-TXN-COLLECT) = (NBR-TXN-COLLECT) + 1;
          DOEND << Get transaction from form and file it >>
      ELSE

      << Prompt the operator to correct errors >>

             PERFORM PROMPT-OPERATOR;
RETURN;

COLLECT-TRANSACTIONS:
<<*****************>>

  << setup form and get transaction entry form >>

  LET (REPEATAPP) = (ZERO);
  LET (FREEZEAPP) = (ZERO);
  MOVE (NFNAME) = "DEDUCTION";
  PROC VGETNEXTFORM((COMAREA));‘‘
  IF (CSTATUS) <> (ZERO) THEN
    DO
    LET (STOP-NOW) = (TRUE);
    MOVE (ERROR-LOCATION) = "**** procedure: Collect Transactions"
                            " - Form Retrieval";
    PERFORM GET-ERROR-MESSAGE;
    RETURN;
    DOEND;

<< Toggle three fields in form to "input allowed" >>

<< Screen order is indicated to field change intrinsic as
   negative number

LET OFFSET(FIELDSPECS) = 0;
LET OFFSET(FIELDENTRY) = 0;
LET (FIELD-ID) = (SCREEN-ORDER) * [-1];
LET (CHANGE-TYPE) = 5;
MOVE (CHANGE-SPEC) = "O";
LET OFFSET(FIELDSPECS) = 8;
LET OFFSET(FIELDENTRY) = 20;
LET (FIELD-ID) = (SCREEN-ORDER) * [-1]
LET (CHANGE-TYPE) = 5;
MOVE (CHANGE-SPEC) = "O";
LET OFFSET(FIELDSPECS) = 16;
LET OFFSET(FIELDENTRY) = 40;
LET (FIELD-ID) = (SCREEN-ORDER) * [-1];
LET (CHANGE-TYPE) = 5;
MOVE (CHANGE-SPEC) = "O";
LET (NUMSPECS) = 3;
PROC VCHANGEFIELD((COMAREA),(FIELDSPECS-ITEM),(NUMSPECS));
IF (CSTATUS) <> (ZERO) THEN
  DO
  LET (STOP-NOW) = (TRUE);
  MOVE (ERROR-LOCATION) = "**** procedure: Collect Transactions"
                          " - Field Type Updates";
  PERFORM GET-ERROR-MESSAGE;
  RETURN;
  DOEND;

<< Load window message >>
<…sc><…x>

<ex><esc>

LET (MSGBUFLEN) = 79;
MOVE (MSGBUF) = "Fill in Deduction Transaction according to "
                "worksheet";
PROC VPUTWINDOW((COMAREA),%(MSGBUF),(MSGBUFLEN));
IF (CSTATUS) <> (ZERO) THEN
   DO
   LET (STOP-NOW) = (TRUE);
   MOVE (ERROR-LOCATION) = "**** procedure: Collect Transactions"
                           " - Window Load";
   PERFORM GET-ERROR-MESSAGE;
   RETURN;
   DOEND;

<< Init form >>
```

```
PROC VINITFORM((COMAREA));
IF (CSTATUS) <> (ZERO) THEN
   DO
   LET (STOP-NOW) = (TRUE);
   MOVE (ERROR-LOCATION) = "**** procedure: Collect Transactions"
                           " - Form Init";
   PERFORM GET-ERROR-MESSAGE;
   RETURN;
   DOEND;

<< Show form >>

PROC VSHOWFORM((COMAREA));
IF (CSTATUS) <> (ZERO) THEN
   DO
   LET (STOP-NOW) = (TRUE);
   MOVE (ERROR-LOCATION) = "**** procedure: Collect Transactions"
                           " - Form Display";
   PERFORM GET-ERROR-MESSAGE;
   RETURN;
   DOEND;

<< Setup and loop on transaction until it can be filed >>

LET (FOUND-DATA-ERRS) = (TRUE);
LEVEL;
   IF (FOUND-DATA-ERRS) = (FALSE) THEN
        END(LEVEL)
   ELSE IF (STOP-NOW) = (TRUE) THEN
        END(LEVEL)
   ELSE IF (DONE-WITH-TRANS) = (TRUE) THEN
        END(LEVEL)
   ELSE
        DO
        PERFORM READ-EDIT-FILE-TRANSACTION;
        END;
        DOEND;
RETURN;

SETUP-FOR-WORK:
<<***********>>

  LET (ZERO) = 0;
  LET (ONE) = 1;
  LET (EIGHT) = 8;
  LET (STOP-NOW) = (FALSE);
  LET (DONE-WITH-TRANS) = (FALSE);
  LET (NBR-TXN-COLLECT) = (ZERO);

  << Init Comarea >>

  << Set language to SPL. This is the default language if TRANSACT

      opens the formfile. However, in this language, all character
      arrays must be passed as byte addresses, ie %(NAME)           >>

  LET (LANGUAGE) = 3;
  LET (COMAREALEN) = 60;
  LET (LABELOPTIONS) = 1;
  MOVE (CFNAME) = " ";
  MOVE (NFNAME) = " ";
  LET (FORMSTORESIZE) = 4;

  << Open of Transaction file done in SYSTEM statement. Opened
      so that new records are appended to those already in the file >>

  << Open forms file >>

  MOVE (FILENAME) = "PAYROLL.VPLUS.MILLER";
  PROC VOPENFORMF((COMAREA),%(FILENAME));
  IF (CSTATUS) <> (ZERO) THEN
    DO
    LET (STOP-NOW) = (TRUE);
    MOVE (ERROR-LOCATION) = "**** procedure: Setup For Work - "
                            "Forms File Open";
    PERFORM GET-ERROR-MESSAGE;
    RETURN;
    DOEND;

  << Open terminal >>

  MOVE (FILENAME) = "HPTERM";
  PROC VOPENTERM((COMAREA),%(FILENAME));
  IF (CSTATUS) <> (ZERO) THEN
    DO
    LET (STOP-NOW) = (TRUE);
    MOVE (ERROR-LOCATION) = "**** procedure: Setup For Work - "
                            "Terminal Open";
    PERFORM GET-ERROR-MESSAGE;
    RETURN;
    DOEND;

  << Translate field names to screen order >>


     << Three of the fields in the form used by this application
          need to be toggled from "display only" to "input allowed".
           In order to do this, we first translate field names to
           screen order.                                          >>

     << Setup to retrieve screen order for three specified fields >>

     MOVE (CFNAME) = "DEDUCTION";
     MOVE (FIELDINFO) = " ";
     LET (NUM-ENTRIES) = 3;
     LET (ENTRY-LEN) = 10;
     MOVE (FORM-NAME) = "DEDUCTION";
     LET OFFSET(FIELDENTRY) = 0;
     MOVE (FIELD-NAME) = "BADGE NUMBER";
     LET OFFSET(FIELDENTRY) = 20;
     MOVE (FIELD-NAME) = "LAST_NAME";
     LET OFFSET(FIELDENTRY) = 40;
     MOVE (FIELD-NAME) = "SUR_NAME";

     << Set length of entire info buffer >>
```

```
LET (INFOBUFLEN) = (NUM-ENTRIES) * (ENTRY-LEN) + 10;
PROC VGETFIELDINFOR((COMAEA),(FIELDINFO),(INFORBUFLEN));
IF (CSTATUS) <> (ZERO) THEN
    DO
    LET (STOP-NOW) = (TRUE);
    MOVE (ERROR-LOCATION) = "**** procedure: Setup For Work - "
                            "Field Info Retrieval";
    PERFORM GET-ERROR-MESSAGE;
    DOEND;
RETURN;
```

PASCAL

```
$TITLE 'VPLUS/V Data Entry Sample Program'$

{

This application collects employee payroll deduction transactions
and places the edited transactions into a file.

For this application:           Enter key = edit and file transaction;

                                       f8 = exit application;

                         all other f keys = redo transaction.

Each transaction entered by the operator is subjected to the data
edits embedded within the input form.

The application continues to collect transactions until either
the operator signals to exit or a system error is detected.
}

PROGRAM Pascal_Sample (output);

CONST
  MAXWINDOWLEN          =  150;
  FILENAMELEN           =  86;

TYPE
  SMALL_INT             =  0..65535

  PAC_ 2                =  PACKED ARRAY [1..2] OF CHAR;
  PAC_ 4                =  PACKED ARRAY [1..4] OF CHAR;
  PAC_ 8                =  PACKED ARRAY [1..8] OF CHAR;
  PAC_ 16               =  PACKED ARRAY (1..16] OF CHAR;
  PAC_ 70               =  PACKED ARRAY [1..70] OF CHAR;
  PAC_ 80               =  PACKED ARRAY [1..80] OF CHAR;
  PAC_ 200              =  PACKED ARRAY [l..200] OF CHAR;
  PAC_ FILENAME         =  PACKED ARRAY [1..FILENAMELEN] OF CHAR;
  PAC_ MAXWINDOWLEN     =  PACKED ARRAY [1..MAXWINDOWLEN]
                           OF CHAR;

  TWO_BYTE_SUB_RANGE  =  PACKED ARRAY [ 1. .2] OF 0. .255;

  COMAREA REC = RECORD
     CSTATUS         : SMALL_INT;
     LANGUAGE        : SMALL_INT;
     COMAREALEN      : SMALL_INT;
     USERBUFLEN      : SMALL_INT;
     CMODE           : SMALL_INT;
     LASTKEY         : SMALL_INT;
     NUMERRS         : SMALL_INT;
     WINDOWENH       : SMALL_INT;
     MULTIUSAGE      : SMALL_INT;
     LABELOPTIONS    : SMALL_INT;
     CFNAME          : PAC_16;
     NFNAME         : PAC_16;
     REPEATAPP      : SMALL_INT;
     FREEZEAPP      : SMALL_INT;
     CFNUMLINES     : SMALL_INT;
     DBUFLEN        : SMALL_INT;
     SKIP_31        : SMALL_INT;
     LOOKAHEAD      : SMALL_INT;
     DELETEFLAG     : SMALL_INT;
     SHOWCONTROL    : SMALL_INT;
     SKIP_35        : SMALL_INT;
     PRINTFILENUM   : SMALL_INT;
     FILERRNUM      : SMALL_INT;
     ERRFILENUM     : SMALL_INT;
     FORMSTORESIZE  : SMALL_INT;
     SKIP_40        : SMALL_INT;
     SKIP_41        : SMALL_INT;
     SKIP_42        : SMALL_INT;
     NUMRECS        : INTEGER;
     RECNUM         : INTEGER;
     SKIP_47        : SMALL_INT;
     SKIP_48        : SMALL_INT;
     TERM_FILEN     : SMALL_INT;
     SKIP 50        : SMALL_INT;
     SKIP_51        : SMALL_INT;
     SKIP_52        : SMALL_INT;
     SKIP_53        : SMALL_INT;
     SKIP_54        : SMALL_INT;
     RETRIES        : SMALL_INT;
     TERM_OPTIONS   : SMALL_INT;
     ENVIRON        : SMALL_INT;
     USERTIME       : SMALL_INT;
     IDENTIFIER     : SMALL_INT;
     LABELINFO      :TWO_BYTE_SUB_RANGE;
       END;

  FIELDENTRY_REC = PACKED RECORD
     FIELD_NAME           : PAC_16;
     SCREEN_ORDER         : SMALL_INT;
       END;


  FIELDINFO REC = PACKED RECORD
     NUM_ENTRIES            : SMALL_INT;
     ENTRY_LEN              : SMALL_INT;
     FORM_NAME              : PAC 16;
     FIEL_ENTRY             : PACKED ARRAY [1..3] OF FIELDENTRY_REC;
       END;


   FIELDSPECS_REC = RECORD
     FIELD_ID             : SMALL_INT;
     CHANGE_TYPE          : SMALL_INT;
     CHANGE_SPEC          : PAC_4;
       END;
CONST
    LABELINFO_INIT = TWO_BYTE_SUB_RANGE [2 OF 0];
```

```
{ Comarea initialization constant record }

  COMAREA_INIT = COMAREA REC
    [ CSTATUS             : 0,
       LANGUAGE           : 5,   {Pascal  }
       COMAREALEN         :60,
       USERBUFLEN         : 0,
       CMODE              : 0,
       LASTKEY            : 0,
       NUMERRS            : 0,
       WINDOWENH          : 0,
       MULTIUSAGE         : 0,
       LABELOPTIONS       : 1,   {activate labels}
       CFNAME             : '',
       NFNAME             : '',
       REPEATAPP          : 0,
       FREEZEAPP          : 0,
       CFNUMLINES         : 0,
       DBUFLEN            : 0,
       SKIP_31            : 0,
       LOOKAHEAD          : 0,   {no form background loading}
       DELETEFLAG         : 0,
       SHOWCONTROL        : 0,
       SKIP_35            : 0,
       PRINTFILENUM       : 0,
       FILERRNUM          : 0,
       ERRFILENUM         : 0,
       FORMSTORESIZE      : 4,   {local form storage}
       SKIP_40            : 0,
       SKIP_41            : 0,
       SKIP_42            : 0,
       NUMRECS            : 0,
       RECNUM             : 0,
       SKIP_47            : 0,
       SKIP_48            : 0,
       TERM_FILEN         : 0,
       SKIP_50            : 0,
       SKIF_51            : 0,
       SKIP_52            : 0,
       SKIP_53            : 0,
       SKIP_54            : 0,
       RETRIES            : 0,
       TERM_OPTIONS       : 0,
       ENVIR0N            : 0,
       USERTIME           : 0,
       IDENTIFIER         : 0,
       LABELINFO          :  LABELINFO_INIT ];
 VAR
   COMAREA                        : COMAREA_REC;
   DATABUF                        : PAC_200;
   DATABUFLEN                     : SMALL_INT;
   DONE_WITH_TRANSACTIONS         : BOOLEAN;

  ERROR_LOCATION                 : PAC_70;
  FIELDINFO                      : FIELDINFO_REC;
  FIELDSPECS                     : ARRAY [1..3] OF FIELDSPECS_REC;
  FILENAME                       : PAC_FILENAME;
  FOUND_DATA_ERRORS              : BOOLEAN;
  INFOBUFLEN                    : SMALL_INT;
  MSGBUF                        : PAC MAXWINDOWLEN;
  MSGBUFLEN                     : SMALL_INT;
  ERRMSGLEN                     : SMALL_INT;
  NBR_TXN_COLLECTED             : INTEGER;
  NUMSPECS                      : SMALL_INT;
  STOP_NOW                      : BOOLEAN;
  TXN_FILE                      : TEXT;


{ Procedure Declarations }

PROCEDURE VCHANGEFIELD              ; INTRINSIC;
PROCEDURE VCLOSEFORMF               ; INTRINSIC;
PROCEDURE VCLOSETERM                ; INTRINSIC;
PROCEDURE VERRMSG                   ; INTRINSIC;
PROCEDURE VFIELDEDITS               ; INTRINSIC;
PROCEDURE VFINISHFORM               ; INTRINSIC;
PROCEDURE VGETBUFFER                ; INTRINSIC;
PROCEDURE VGETFIELDINFO             ; INTRINSIC;
PROCEDURE VGETNEXTFORM              ; INTRINSIC;
PROCEDURE VINITFORM                 ; INTRINSIC;
PROCEDURE VOPENFORMF                ; INTRINSIC;
PROCEDURE VOPENTERM                 ; INTRINSIC;
PROCEDURE VPUTWINDOW                ; INTRINSIC;
PROCEDURE VREADFIELDS               ; INTRINSIC;
PROCEDURE VSHOWFORM                 ; INTRINSIC;

PROCEDURE DISPLAY_SYSTEM_ERROR;

BEGIN

WRITELN ('**** Transaction collection facility detected system ',
         'error at: ',
         ERROR_LOCATION);

WRITELN ('**** The error messaged returned is:');
WRITELN ('**** ', MSGBUF : MSGBUFLEN);

END;


PROCEDURE CLEANUP_AFTER_WORK;

{ Note that this procedure unconditionally attempts to close
  the forms file and terminal                                }

BEGIN

CLOSE (TXN_FILE);

COMAREA.CSTATUS := 0;

VCLOSEFORMF (COMAREA);

COMAREA.CSTATUS := 0;

VCLOSETERM (COMAREA);
```

```
END;


PROCEDURE GET_ERROR_MESSAGE;

BEGIN

MSGBUFLEN := MAXWINDOWLEN;

VERRMSG (COMAREA,
         MSGBUF,
         MSGBUFLEN,
         ERRMSGLEN);

END;


PROCEDURE PROMPT_OPERATOR;

BEGIN

GET_ERROR_MESSAGE;

VPUTWINDOW (COMAREA,
            MSGBUF,
            ERRMSGLEN);

IF COMAREA.CSTATUS <> 0 THEN
        BEGIN
        STOP_ NOW := TRUE;
        ERROR_LOCATION
           := '**** procedure: Prompt Operator - Window Load';
        GET ERROR_MESSAGE;
        END;

IF NOT (STOP_NOW) THEN
        BEGIN
    { Display update }

    VSHOWFORM {COMAREA};

    IF COMAREA.CSTATUS <> 0 THEN
         BEGIN
         STOP_NOW := TRUE;
         ERROR_LOCATION
            : = '**** procedure: Collect Transactions - Display Update';
         GET_ERROR_MESSAGE;
         END;

    END; { Display update }

END; {Procedure Prompt Operator }


 WHILE         (FOUND_DATA_ERRORS)
      AND   (NOT (STOP_NOW))
      AND   (NOT (DONE_WITH_TRANSACTIONS)) DO
        READ_EDIT_FILE_TRANSACTION;
```

```
   END;

END; { Procedure Collect Transactions }

PROCEDURE SETUP_FOR_WORK;

BEGIN

{ Init Comarea }

COMAREA := COMAREA_INIT;

{ Open Transaction File so that new transactions are
  added to those already in the file                  }

APPEND (TXN_FILE, 'PAYTXN');

{ Open forms file }

FILENAME := 'PAYROLL.WORK.ADMIN';

VOPENFORMF (COMAREA,
            FILENAME);

IF COMAREA.CSTATUS <> 0 THEN
   BEGIN
   STOP_NOW := TRUE;
   ERROR_LOCATION
        := '**** procedure: Setup For Work - Forms File Open';
   GET_ERROR_MESSAGE;
   END;

IF NOT (STOP_NOW) THEN
   BEGIN

   { Open terminal }

   FILENAME := 'HPTERM';

   VOPENTERM (COMAREA,
              FILENAME);

   IF COMAREA.CSTATUS <> 0 THEN
        BEGIN
        STOP_NOW := TRUE;
        ERROR_LOCATION
          := '**** procedure: Setup For Work - Terminal Open';
        GET_ERROR_MESSAGE;
        END;


    END; { Open terminal }
    IF NOT (STOP_NOW) THEN
    BEGIN

    { Translate field names to screen orders }
    { Three of the fields in the form used by this application
      need to be toggled from "display only" to "input allowed".
      In order to do this, we first translate field names to
      screen orders.                                           }

    { Setup to retrieve screen order for three specified fields }

    FIELDINFO.NUM_ENTRIES := 3;

    FIELDINFO.ENTRY_LEN   := 9; { Field name key and screen order }

    FIELDINFO.FORM_NAME := 'DEDUCTION';

    FIELDINFO.FIELDENTRY [1].FIELD_NAME   := 'BADGE_NUMBER';
    FIELDINFO.FIELDENTRY [1].SCREEN_ORDER := 8224;  {ASCII blanks}

    FIELDINFO.FIELDENTRY [2].FIELD_NAME   := 'LAST_NAME';
    FIELDINFO.FIELDENTRY [2].SCREEN_ORDER := 8224;  {ASCII blanks}

    FIELDINFO.FIELDENTRY [3].FIELD_NAME   := 'SUR_NAME';
    FIELDINFO.FIELDENTRY [3].SCREEN_ORDER := 8224;  {ASCII blanks}

    { Set length of entire info buffer }

    INFOBUFLEN
       := (FIELDINFO.NUM_ENTRIES * FIELDINFO.ENTRY_LEN) + 10;

    VGETFIELDINFO (COMAREA,
                   FIELDINFO,
                   INFOBUFLEN);

    IF COMAREA.CSTATUS <> 0 THEN
       BEGIN
       STOP_NOW := TRUE;
       ERROR_LOCATION
          : = '**** procedure: Setup For Work - Field Info Retrieval';
       GET_ERROR_MESSAGE;
       END;

    END; { Translate field names to screen orders }

 END; { Procedure Setup For Work }

 BEGIN

 { Sample program outer block }

STOP_NOW                 := FALSE;
DONE_WITH_TRANSACTIONS := FALSE;

NBR_TXN_COLLECTED := 0;

SETUP_FOR_WORK;

WHILE (NOT (STOP_NOW))
  AND (NOT (DONE_WITH_TRANSACTIONS)) DO
      COLLECT_TRANSACTIONS;
```

```
CLEANUP_AFTER_WORK;

WRITELN ('Deduction transactions collected this session = ',
           NBR_TXN_COLLECTED);

IF STOP NOW THEN
     DISPLAY_SYSTEM_ERROR;

END.
```

B VPLUS Error Messages

Note the following words or characters within messages:

This appendix lists all error messages that can occur during execution of VPLUS in interactive mode or batch mode. The messages consist of the error number and the associated message. Cause and action text is provided for most messages.

- FSERR

  Indicates a file system error has occurred. When the number following FSERR is negative, refer to the appropriate group (FORMSPEC, REFSPEC, etc.) of "Access Method Errors" listed in this appendix.

- COM 'STATUS

  Indicates an error executing one of the VPLUS intrinsics.

- Internal

  Indicates that the problem is internal to the system; the user cannot easily correct the problem. If you encounter a message with this condition, contact the system manager for advice.

- Exclamation point ()

  Exclamation points in the message are replaced by particular numbers or characters before the message is displayed. For instance, if a message contains FSERR, the exclamation marks will be replaced by a file error number when the message is displayed.

Classification of Error Messages

VPLUS issues the following basic types of error messages:

- FORMSPEC
- VPLUS Intrinsics
- SNA DHCF
- REFSPEC
- REFORMAT

The following tables provide a map of the classification of messages within each category, the number range for the classification, and the associated page numbers where error information can be found.

FORMSPEC Error Messages

Access Method Messages

Error	Message	Cause/Action
086	`Access method: file code error.`	
087	`Access method: Attempt to add a duplicate key.`	
088	`Access method: Internal error. Key insertion location error.`	
089	`Access method: Internal error. Block not present.`	
090	`Access method: Attempt to open an old file as new.`	
091	`Access method: Internal error. Key block buffer error,`	
092	`Access method: Internal error. Invalid key block number.`	
093	`Access method: Internal error. Block not verified.`	
094	`Access method: Record not found.`	
095	`Access method: Not enough space in extension for the directory.`	Increase the *MAXDATA* parameter on the PREP or RUN command. Check the FORMSPEC listing for directory size.
096	`Access method: Internal error. Parent block not found.`	
097	`Access method: Internal error. Illegal entry number.`	
098	`Access method: The file is not a KSAMless forms/ ref file.`	
099	`Access method: The file is at EOF.`	

Screen Definition Messages

Error	Message	Cause/Action
101	`Form is too big, overflowed the screen buffer.`	Maximum screen buffer size is 7984 bytes. Reduce the size of your form.

Error	Message	Cause/Action
102	`Form ! is too big to becompiled.`	Form code record has exceeded 8K bytes. Maximum number of fields is 127. Reduce the number of fields, size of screen, or amount of processing specifications.
103	`An END FIELD must have a matching START FIELD.`	Check your screen design to make sure that each "end field" delimiter has a matching "begin field" delimiter. If your delimiters are the unprinted kind (ESC [and ESC]), it may be more convenient to delete the line containing the delimiters and to replace them.
106	`Internal error: All of the screen was not read correctly.`	Possibly a missing record terminator. Press **ENTER** again. If the error still occurs, reduce the size of the form or of the processing specifications.
107	`A name cannot be longer than 15 characters.`	Limit your form, field, or save field names to 15 characters.
108	`A name cannot have embedded " "'s or "."'s.`	Blanks and periods not allowed in names.
109	`A name must start with an alphabetic character.`	Special characters and digits are not allowed.
110	`A name cannot have embedded " "'s or "."'s.`	See error 108.
111	`A name can only contain alphas, digits and "_"'s.`	Letters (A-Z), (a-z), numbers (0-9), or an underline (_) are allowed in a name.
112	`Internal error: Unexpected CR.`	Unexpected carriage return.
113	`Internal error: Screen terminator never found.`	See message for error 106. Also make sure enough terminal buffers are configured to complete the rest of the screen. The number of buffers can be increased to 255.
114	`Internal error: Expected LF after CR.`	Line feed expected after carriage return.
115	`A field must contain a name.`	Each field on the screen must have a name within the field boundaries.
116	`Internal error writing screen message. (COM'STATUS !!!!)`	
117	`Terminal error initializing new screen. (COM'STATUS !!!!)`	
118	`Terminal error writing new screen. (COM'STATUS !!!!)`	
119	`Terminal error writing new screen. (COM'STATUS !!!!)`	

Error	Message	Cause/Action
120	`Terminal error initializing old screen. (COM'STATUS !!!!)`	
121	`Terminal error writing old screen. (COM'STATUS !!!!)`	
122	`Terminal error reading screen. (COM'STATUS !!!!)`	See error 106.
123	`Error writing the screen source. File full? (FSERR!!!!)`	Could not write the screen source record to the form file. First try to correct the problem using the information given in the File System error message. If you are unsuccessful, try the following: a) The file may be full. Build a larger file using the :`BUILD` command and `FCOPY>` b) A system crash or abort may have corrupted the form. Re-enter all menus defining the form. c) If the above intrinsics fail, delete the form and recreate it.
124	`Internal error reading the screen source for form !. (FSERR !!!)`	Could not read the screen source record from the forms file. See error 123.
125	`A name can only be in the first line of the field.`	If a field is continued on a second line, the name must be in the first line.
126	`A name can only be in the first line of the field.`	See error 125.
127	`Internal error: Allocate of GLOB'SCREEN failed.`	
128	`Error writing the global screen record. File full? (FSERR!!!!)`	See error 123.
129	`Internal error: Allocate of STBL failed.`	Could not allocate stack space for symbol table.
130	`Error writing STBL to the forms file. File full? (FSERR!!!!)`	See error 123.
131	`INTERNAL ERROR: changing screen of family.`	
132	`Only 128 fields are allowed in a form.`	Reduce the number of fields in the form.

Field Symbol Table Messages

Error	Message	Cause/Action
301	`Internal error: FST ALLOCATE for form ! failed.`	FST is Field Symbol Table. See error 129.
302	`Internal error: Field symbol table for form ! is missing.`	See error 123. Could not find Field Symbol Table for old form in form file. Press **ENTER** on screen design to re-create FST.
303	`Error writing FST to forms file. File full?`	Could not write FST record in forms file. See error 123.
304	`Internal error: FST does not have any more room.`	Renumber the form in batch mode.
305	`The name must be unique for this form.`	The field name is the same as another field name in the same form.
306	`Internal error: FST search range error.`	Field Symbol Table search problem. See error 304.
307	`Internal error: FST field length changed.`	Field Symbol Table access problem. See error 304.
308	`Internal error: FST start order.`	Field Symbol Table structure problem. See error 304.
309	`Internal error: FST end order.`	Field Symbol Table structure problem. See error 304.
313	`Internal error: FST unreferenced error.`	Field Symbol Table structure problem. See error 304.
314	`Internal error: In form !, field ! has an invalid screen order.`	Field Symbol Table structure problem. See error 304.
315	`The field could not be found.`	Field name does not exist.
316	`Internal error: FST update screen order.`	See error 308.
317	`The field number must be unique for this form.`	See error 308.
319	`In form !, screen order for field ! is out of bounds.`	See error 304.
320	`Internal error: FST wrap loop.`	See error 308.
321	`Internal error reading FST for form !. (FSERR !!!)`	See errors 123, 303.
322	`Internal error storing FST for form !. (FSERR !!!)`	See errors 123, 303.
	`Internal error: CFG ALLOCATE for form ! failed.`	

Error	Message	Cause/Action
	`The field number cannot be more than 256.`	Too many modifications have been made to the screen for this form. The form must be rebuilt from scratch or renumbered in batch.

Menu Processing Utility Messages

Error	Message	Cause/Action
601	`Must be B, H, I, U, 1-8, in any combination, or NONE.`	Enhancement code is incorrect; check and correct.
604	`The name cannot have embedded blanks.`	A field or form name has embedded blank; delete blank or use underline (_).
605	`The form name must be unique for this forms file.`	Form name duplicates another form name in file; make name unique.
606	`Must be NUMn, DIG, IMPn, MDY, DMY, YMD, or CHAR.`	Data type incorrect; use one of the listed codes - n indicates number of decimal positions.
608	`Must be O (optional), R (required), P (process) or D (display only).`	Field type incorrect; use one of the listed codes.
609	`Internal error: Global allocate failed.`	
610	`Error writing globals buffer. File full? (FSERR!!!!)`	See error 123.
611	`For a NUMn data type, n must be a digit.`	Use digit (0-9) after NUM data type to indicate number of decimal positions.
612	`For the IMPn data type, n is required.`	You must specify the number of implied decimal places in an IMP type field.
613	`The name cannot be a reserved word.`	Refer to the list of the FORMSPEC reserved words.
615	`Color pair value must be between 1 and 8.`	

Menu Init and Processing Messages

Error	Message	Cause/Action
701	`The name cannot be blank.`	The form name must be specified on FORM menu.
702	`The next form name cannot be blank.`	Specify a particular next form name, or `$HEAD, $END, $RETURN`, or `$REFRESH`.
703	`Error trying to open the forms file. (FSERR!!!!)`	Refer to the File System Error number for the cause of the open failure. Check form name. Do not use the key file if the form file is old.
704	`The window line must be a number between 0 and 24, inclusive.`	Correct window specification on GLOBALS menu.

Error	Message	Cause/Action
705	`Must be N (no repeat), R (repeat), or A (repeat and append).`	Correct Repeat Option on FORMS menu.
706	`Must be C (clear), A (append), or F (freeze, then append).`	Correct Next Form option on FORMS menu.
707	`Internal error writing top lines to terminal.then append).`	Field Menu.
708	`Internal error writing terminal initialization.(COM’STATUS !!!!)`	
709	`Internal error writing field specs to terminal. (COM’STATUS !!!!)`	Field Menu.
710	`Error in VREADFIELDS. (COM’STATUS !!!!)`	
711	`Internal error reading the terminal, unformatted. (COM’STATUS !!!!)`	Screen read error or processing specification read error.
712	`Internal error positioning the cursor. (COM’STATUS !!!!)`	
713	`Error in VSHOWFORM. (COM’STATUS !!!!)`	
714	`Internal error writing the window. (COM’STATUS !!!!)`	
715	`Internal error writing the Field Menu source record.(FSERR!!!!)`	
716	`Internal error writing the error cursor pos’ n. (COM’STATUS !!!!)`	
717	`Internal error writing the field top lines. (COM’STATUS !!!!)`	Field Menu.
718	`Error in VGETNEXTFORM for Main Menu. (COM’STATUS !!!!)`	
719	`Error in VSHOWFORM for Main Menu. (COM’STATUS !!!!)`	
720	`Error in VREADFIELDS for Main Menu. (COM’STATUS !!!!)`	
721	`Selection cannot be blank.`	Enter a value in the MAIN menu selection box.
722	`The form could not be found.`	Form specified on MAIN menu not found in forms file. In batch mode, form not found in current forms file.

Error	Message	Cause/Action
723	`Selection must be one of the characters listed below.`	Enter one of the specified characters in the MAIN menu selection box.
724	`This selection requires a form name to be supplied.`	Specify a form name in the appropriate box, or enter a different character in the selection box.
725	`Err writing form int'm. rec for form !. File Full? (FSERR !!!)`	See error 123. Field Symbol Table error.
726	`Internal error initializing the form!.`	
727	`A forms file has not yet been specified, so selection is invalid.`	To copy or compile from MAIN menu, forms file must have been specified on FORMS FILE menu.
728	`File does not exist. Press ENTER to create file.`	
734	`There was an error trying to open this forms file. (FSERR !!!!)`	
736	`Internal error reading the forms file. (FSERR !!!!)`	
740	`There are no forms in this forms file.`	Specified forms file has no defined forms.
742	`Save field length cannot be blank.`	Enter number of characters in save field on SAVE FIELD menu.
743	`Save field length must be an integer greater than 0.`	
744	`This save field already exists.`	Use a new save field name.
745	`A forms file can only have 20 save fields.`	
746	`There are no save fields in this forms file.`	
747	`The save field name cannot be blank.`	Enter a save field name on the SAVE FIELD menu.
748	`WARNING: Forms file modified and not compiled.Press EXIT to exit.`	You modified the forms file and then pressed **EXIT**. If you want to compile before terminating FORMSPEC, press MAIN, otherwise press **EXIT** again.
749	`This file is a Fast Forms File and cannot be modified.`	You are attempting to modify a fast forms file; use the forms file name you originally specified.
750	`This file is a Reformat file; use REFSPEC to modify it.`	Specify a forms file name, or `RUN REFSPEC.PUB.SYS` to modify the reformat file.

Error	Message	Cause/Action
751	`This file is not a VPLUS Forms File.`	You specified an incorrect file name.
752	`Error opening the Fast Forms File. (FSERR !!!!)`	
753	`Error reading the forms file. (FSERR !!!!)`	
754	`Error writing to the Fast Forms File. File full? (FSERR !!!!)`	See error 123.
755	`Error finding the correct record. (FSERR !!!!)`	
756	`Error reading the chained records. (FSERR !!!!)`	
757	`Error opening the old Fast Forms File. (FSERR !!!!)`	
758	`Error purging the old Fast Forms File. (FSERR !!!!)`	
759	`Error closing the Fast Forms File. (FSERR !!!!)`	
760	`The old file is not a VPLUS Fast Forms File.`	The fast forms file name you specified is not an existing VPLUS fast forms file.
761	`This file is not a VPLUS forms file.`	See error 751.
762	`Fast forms file name must be specified if key file is specified.`	You are trying to compile to a fast forms file without naming the data file.
763	`Copy/rename/delete I/O error. (FSERR !!!!)`	See error 123.
764	`Error when deleting an old record.`	See error 123.
765	`The specified family is not defined.`	
766	`Cannot change screen, the form is a family member.`	User pressed **ENTER** on the screen of a child form or attempted to change the screen. Press **REFRESH** (f4) to repaint the original and press **NEXT** (f6) to go to the field menus.
767	`Cannot change family of the form.`	Do not attempt to change the name in the "Reproduced from" field of the form menu.
768	`List operation failed.`	A write to file FORMLIST failed. Check for a valid file equation or for spooling problems.

Batch Mode Messages

Error	Message	Cause/Action
769	`Invalid batch mode command.`	FORMSPEC expected a valid batch mode command or parameter.
770	`Warning: Forms file modified and not compiled.`	The forms file must be compiled before it can be used.
771	`Could not open command and not compiled.`	Check file name, security, or if in use elsewhere.
772	`Could not open list file.`	Check file name, security, or if in use elsewhere.
774	`In form !, field !, "!" is undefined:`	An error was found in a field during compilation. (See next error message printed.)
775	`Internal error in form !, field !:`	An error was found in a field during compilation. (See next error message printed.)
776	`Form !, field ! has semantic error:`	An error was found in a field during compilation. The syntax of the statement in the field may be correct, but its meaning is wrong. (See next error message printed.)
777	`Form !, field ! has a data type error:`	An error was found in a field during compilation. Data type inconsistent with specification. (See next error message printed.)
778	`Form !, field ! hasa syntactic error:`	Statement syntax is incorrect. (See next error message printed.)
780	`Commands must be preceded by a FILE command.`	Current forms file must be specified.
781	`Missing form name.`	Form name must be specified.
782	`Missing forms file name.`	Forms file name must be specified.
783	`Missing fast forms file name.`	Fast forms file name must be specified when "INTO" is used.
784	`Too many arguments.`	Too many parameters were specified for the command.
785	`Expecting "TO form-name".`	The form name must be specified when "TO" is used.
786	`Expecting "IN forms-file" or "TO form-name".`	`COPY` command must include either "IN file" or "TO newform".
787	`Expecting "INTO fast-forms-file".`	The only option for the `COMPILE` command is "INTO" followed by the fast forms file name.
788	`Internal error in FORMSPEC processing.`	See System Manager.

Error	Message	Cause/Action
789	`All commands ignored until next FILE command.`	Either a command was not recognized or an error occurred with a `FILE` command.
790	`Compile failed.`	

RELATE Command Messages

Error	Messages	Cause/Action
791	`Child cannot have the same name as parent.`	Same form name was specified as the parent and child form.
792	`Form is already child of that parent.`	The forms are already related.
793	`Child form is already related to another parent.`	A child cannot have more than one parent.
794	`Form specified as child cannot have any children.`	A child form cannot be a parent.
795	`Form layout of child must be the same as parent.`	The form layouts must be identical, including field tags.
796	`Field numbers of child must be the same as parent.`	When the field numbers do not match, the forms cannot be related.
797	`Correct syntax is: "RELATE child-form-name TO parent-form-name".`	Arguments in the batch mode command were mistyped.
798	`KSAM forms files can no longer be modified - use CONVERT. PUB.SYS.`	Refer to Appendix H for instructions.
799	`Cannot change color forms file on a non-color terminal.`	
800	`Internal error in color conversion.`	

Menu Controller Messages

Error	Messages	Cause/Action
802	`The PREV key is not appropriate; there are no previous menus.`	You are currently at the first menu; press **NEXT** to go to the next menu, **NEXT FORM** to display the next form menu, or use the MAIN menu to select a particular menu.
803	`The function key just hit is not defined for this mode.`	
804	`Internal error reading source record from forms file. (FSERR !!!!)`	See error 123
805	`Internal error getting next form !. (COM'STATUS !!!)`	
806	`Error in VSHOWFORM. (COM'STATUS !!!!)`	
807	`Error in VREADFIELDS. (COM'STATUS !!!!)`	

Error	Messages	Cause/Action
808	`Error in VGETBUF. (COM’STATUS !!!!)`	
809	`Error writing source to forms file. File full? (FSERR !!!!)`	See error 123.
811	`Internal error reading form record for form !. (FSERR !!!)`	See error 123.
812	`Internal error reading globals record. (FSERR !!!!)`	See error 123.
813	`Internal error allocating GLOB’SCREEN for form !.`	
814	`Internal error allocating STBL for form !.`	
815	`NEXT is not meaningful until a forms file name is specified.`	Enter a forms file name on the FORMS FILE menu. (Exit if necessary and run FORMSPEC again to display the menu.)
816	`NEXT is not appropriate, since there are no more forms.`	Use **PREV** or **PREV FORM** key, or go to the MAIN menu to request a particular form.
817	`The save field could not found.`	
820	`The PREV FORM key is not appropriate; there are no previous forms.`	Use **NEXT** or **NEXT FORM** key, or go to the MAIN menu to request a particular form.
821	`Internal error allocating the globals buffer.`	
822	`This value is required and cannot be blank.`	
823	`Enter "Y" to go to function key labels menu.`	To define global function key labels, enter "Y" on the last line of the GLOBALS menu. To define form function key labels, enter "Y" on the Function Key Label line of the FORM menu.
824	`Pause interval may not be negative.`	The Split Message Pause must be between 1 and 99.
825	`Invalid error light (A ... P, @@).`	Enter a valid error light (a letter of the alphabet or the @@ character).
826	`Invalid Multifunction reader definition (MARKS or HOLES).`	Enter either of the valid Multifunction reader definitions.
827	`Field may only have the values "YES" or "NO".`	The valid options for Corner Cut Required are YES or NO.
828	`Illegal combination of options for the Multifunction reader.`	The combinations HOLES with Clock On and MARKS with NONE are not valid.

Error	Messages	Cause/Action
829	`Invalid Multifunction reader definition (COD, CAD, NONE).`	Enter one of the valid Multifunction reader definitions.
830	`Invalid barcode type (UPC, EAN, I25, I39, MAT, ILV).`	Enter one of the valid Barcode Reader types.
831	`Internal error: Writing device intermediate record (FSERR !!!!)`	
832	`Invalid combination of PAUSE INTERVAL and WAIT FOR USER`	If Split Message Pause is specified, Wait for User must be NO.
833	`Enter "CONTINUE" only, or press a function key.`	Only CONTINUE may be entered in this field.
834	`NEXT is not appropriate until the forms file has been adjusted.`	The specified forms file is an old version and must be adjusted before going further.
835	`Place name of form to be listed within List Form field.`	
836	`Enter "Y" or "N" only.`	
837	`Enter ‘‘Y’’ to go to Data Type Conversion Menu.`	

Init and Compile Messages

Error	Message	Cause/Action
902	`For form !, the next form ! has not been defined.`	The next form name specified does not exist in the forms file. Change the next form name or create a form with that name.
903	`The head form ! has not been defined.`	The head form specified on the globals menu does not exist in the forms file. Create a form with the head form name or change the head form on the globals menu.
904	`Internal error closing FORMSPEC forms file. (COM'STATUS !!!!)`	
905	`Internal error closing the terminal file. (COM'STATUS !!!!)`	
909	`Error writing the global code record. File full? (FSERR !!!!)`	See error 123.
911	`Error writing form code record for form !. File full? (FSERR !!!)`	See error 123.
912	`Internal error reading screen source for form !. (FSERR !!!)`	
913	`Internal error closing your forms file. (FSERR !!!!)`	
915	`Initial value contains system constant,`	
916	`In form !, the field name ! is not unique.`	Duplicate field name found in indicated form; use FIELD menu to change field name.
917	`Error writing field names table for form !. File full? (FSERR !!!)`	
918	`Error writing save field table for form !. File full? (FSERR !!!)`	See error 123.
919	`The initial value is too long.`	The initial value is longer than the length of the field as defined on the screen menu. Use the screen menu to lengthen the field or shorten the initial value on the field menu.
920	`The initial value contains an invalid DIG value.`	A DIG type field may contain only digits (0-9), no decimal point, commas, or sign.
921	`The initial value contains an invalid NUM value.`	A NUM type field may contain only digits, decimal point, a sign.
922	`The initial value contains an invalid numeric value.`	This field may only have a numeric value; no letters or special characters.

Error	Message	Cause/Action
923	`Internal error: For form !, allocating local buffer for field !.`	
924	`Internal error: For form !, reading source for field !. (FSERR !!)`	See error 123.
925	`In form !, field ! has an invalid initial value.`	The initial value specified does not match the field data type.
926	`In form !, field name !is a reserved word.`	Check list of reserved words and change field name.
927	`The initial value contains an invalid MDY date value.`	Make sure date is in order month day year, is a valid date, spelled correctly.
928	`The initial value contains an invalid DMY date value.`	Make sure date is in order day month year, is a valid date, spelled correctly.
929	`The initial value contains an invalid YMD date value.`	Make sure date is in order year month day, is a valid date, spelled correctly.
932	`The initial value contains an invalid IMP value.`	An implied decimal field may contain only digits, decimal point, commas, sign.
950	`EDIT statements are not allowed in the INIT phase.`	Field edits cannot be performed in the INIT phase; move statement to FIELD phase.
951	`EDIT statements are not allowed in the FINISH phase.`	Field edits cannot be performed in the FINISH phase; move statement to FIELD phase.
952	`Only a field, save field, or implied current field is allowed.`	
953	`Only one character is allowed in a FILL statement.`	Enter single character surrounded by quotes.
954	`Only 10 or 11 is allowed in a CDIGIT statement.`	CDIGIT tests only modulo 10 or modulo 11 check digits.
994	`In form !, field !, "!" is undefined; press NEXT to see error.`	Indicated specification is not recognizable; **NEXT** will display FIELD menu so you can make correction and then press **ENTER**.
995	`Internal error in form !, field ! ; press NEXT to see error.`	**NEXT** displays FIELD menu and another message with the cursor positioned to the location of the error so you can correct the specification and then press **ENTER**.
996	`Form !, field ! has a semantic error; press NEXT to see error.`	**NEXT** displays FIELD menu so you can correct the error and then **NEXT** to see error. press **ENTER**. The syntax of your statement may be correct, but its meaning is in error.

Error	Message	Cause/Action
997	`Form !, field ! has a data type error; press NEXT to see error.`	Data type inconsistent with specification; **NEXT** displays **NEXT** to see error. FIELD menu so you can correct data type or specification and press **ENTER**.
998	`Form !, field ! has a syntactic error; press NEXT to see error.`	**NEXT** displays FIELD menu so you can correct statement syntax and then press **ENTER**.
999	`Field processing specification syntax error.`	Correct syntax of statement and then press **ENTER**. error.

Scanner Messages

Error	Messages	Cause/Action
1001	Too many levels of indenting, limit is !!!!.	Only 10 levels of nesting are allowed.
1002	Improper indenting, statement groups must be vertically aligned.	IF and ELSE parts at same nesting level must be indented vertically aligned the same amount. Maximum levels of nesting for IF is eight levels.
1003	Improper indenting, must be within outer level of indenting.	An ELSE or subsequent IF starts to the left of first IF statement.
1010	More left braces than right braces were found.	Braces must match; add right brace, or delete extra left brace.
1011	More right braces than left braces were found.	Braces must match; add left brace, or delete extra right brace.
1030	Token is too long, limit is !!!! characters.	A string exceeds 120 characters. Correct and press enter.
1031	A $ may only appear with a system defined name, like $empty.	A $ cannot be part of a name unless it is one of the system defined names (see TABLE 03-01).
1032	Expected a statement beginning.	Syntax error. The cursor is positioned to the location of the error.
1033	String is too long, limit is !!!! characters.	
1034	String must end with a quote.	A literal string (character constant) must be enclosed within quotes.
1035	String must begin with a quote.	When continuing a string to the next line, close the quotes, put an & at the end of first line, enclose string on second line within quotes.
1036	Only a comment may appear on the same line after an ampersand.	An ampersand (&) is used to continue a string; it cannot be followed by another statement, only a comment starting with a \.
1037	A date constant must end with an exclamation point.	A date used as a constant must be enclosed within exclamation points.
1038	Date constant is too long, limit is !!!! characters.	
1039	Date constant must be a valid MDY date.	Date constants must be in the order: month day year, and must be valid dates.

Parser Messages

Error	Messages	Cause/Action
1094	`Invalid name: no field or save field with this name is defined.`	A valid field name is expected
1095	`Internal error.`	
1096	`Semantic error.`	See error 1098.
1097	`Data type error: Data types must be compatible.`	Data type and value in processing specification do not correspond.
1098	`Field processing specification syntax error.`	Statement is incorrect. Cursor is positioned to the location of the error. Correct and press enter.

Apply Errors

Error	Message	Cause/Action
1101	`Left operand must be type DIG, NUMn, IMPn, or an expression.`	An arithmetic operand is expected as destination of SET or comparison or an expression. statement. Cursor is positioned to the location of the error.
1102	`Both operands must be type DIG, NUMn, IMPn, or an expression.`	Arithmetic operands expected in SET or comparison statement. Cursor is positioned to the location of the error.
1103	`Operand must be type DIG, NUMn, IMPn, or an expression.`	Arithmetic operand expected in SET or comparison statement.
1104	`Operand data types must be compatible.`	Operands of the same data type are expected in a SET or comparison statement.
1105	`Operand types must agree unless source is a field or a save field.`	Operands must be the same data type unless the source is a field or save field name.
1106	`Right operand must be type CHAR.`	A character type value is expected as source of SET or comparison statement.
1107	`Left operand must be a field or a save field.`	Destination operand of SET statement may only be a field or save field name.
1108	`NFORM name must be a FORM name in quotes, or a type CHAR field.`	In `CHANGE NFORM` statement, the next form is identified by its name in quotes, by a char type field containing the form name, or by a system constant such as $HEAD.
1109	`MINLEN operand must be type DIG, NUMn, IMPn, or a number.`	The minimum field length is specified as a positive integer, or a name of a numeric field.

Error	Message	Cause/Action
1110	`CDIGIT operand must be 10 or 11.`	CDIGIT checks only modulo 10 or 11 check digits.
1120	`Too many statements, code for entire form is too big.`	At compilation, the form information generated does not fit into the maximum record size of 12000 bytes. Reduce the number of fields or processing specifications.
1130	`Improper indenting, statement groups must be vertically aligned.`	Corresponding IF and ELSE parts of IF statement must start in same column.
1131	`Syntax error: A name is not expected here.`	A name occurs where a reserved word is expected. Check spelling, correct and press **ENTER**.
1132	`Syntax error: The statement is incomplete.`	A valid statement ends unexpectedly.Correct and press **ENTER**.
1133	`Syntax error: Unexpected symbol.`	A special character is not expected here.
1923	`Internal error: Unexpected auto-indent.`	Internal error in `IF-THEN-ELSE` statement.
1924	`Internal error: Error numbers are inconsistent.`	
1925	`Internal error: MEM size is inconsistent with ARRAYSIZE.`	
1926	`Internal error: XEVAL recursion underflow.`	
1927	`Internal error: Invalid node in XEVAL.`	
1928	`Internal error: Invalid MATCH operand in XEVAL.`	
1929	`Internal error: Date conversion failed in XEVAL.`	
1930	`Internal error: XEVAL recursion overflow`	
1931	`Internal error: INEXT conversion failed.`	
1932	`Internal error: Operand must be character string constant.`	
1933	`Internal error: Invalid operand in APPLY.`	
1934	`Internal error: Invalid operand in type check.`	

Error	Message	Cause/Action
1935	`Internal error: Invalid operand in MATCH.`	
1936	`Internal error: Invalid operand in CDIGIT.`	
1937	`Internal error: Invalid right operand.`	
1938	`Internal error: Invalid left operand.`	
1939	`Internal error: Invalid source operand.`	
1940	`Internal error: Invalid Dtype in type checking.`	
1941	`Internal error: EXTIN conversion failed.`	
1942	`Internal error: Bad id number.`	
1943	`Internal error: Bad screen order number.`	
1944	`Internal error: MEM dispose failed.`	
1945	`Internal error: Lexic level underflow.`	
1946	`Internal error: Invalid leading blanks.`	
1947	`Internal error: Invalid Dtype in scanner.`	
1948	`Internal error: MEM allocate failed.`	
1949	`Internal error: Invalid production.`	
1950	`Internal error: Scanner error.`	
1951	`Invalid local edit combination.`	An invalid combination of local edits has been specified in the CONFIG phase of the field processing specifications.

Pattern Compile Messages

Error	Message	Cause/Action
2101	`A "(", ")", "[" , or "] is not paired in pattern.`	MATCH statement contains odd number of parentheses or brackets.
2102	`Invalid character found in pattern.`	A character in the expression is not recognized as a valid character in the pattern.
2104	`End of pattern found prematurely.`	Pattern characters were found after the end of the pattern was found.
2105	`Second character in range is lower than first.`	A range (a:b) is transposed, or else a colon was not preceded by an exclamation point.
2203	`Pattern is too big.`	Either split the pattern into smaller patterns (with IF statements), or simplify the pattern.
2900	`Internal error :COMPILE’PATTERN failed.`	

Release Messages

Error	Message	Cause/Action
3097	`Invalid data type conversion notation.`	Enter the correct data type to which the specified data type should be converted.
3098	`Data Type Conversion Record has not been defined.`	Use the Data Type Conversions Menu to set the default data type conversions.
3100	`Internal error: Could not create list process.`	
3101	`Internal error: Could not send mail to list process.`	
3102	`Internal error: Mail from list was lot.`	
3104	`Formsfile reopen failed.Press PREV to reopen.(FSERR !!!!)`	
3105	`Internal error: List could not close the formsfile.`	
3106	`Internal error: List could not close the formsfile.`	
3107	`Internal error: could not flush current form to the forms file.`	
3108	`Internal error: could not flush the field table to the forms file.`	

Error	Message	Cause/Action
3109	`Internal error: could not flush the globals to the forms file.`	
3110	`Internal error: could not close the formsfile.`	
3111	`List could not open the FORMSPEC formsfile. (FSERR !!!!).`	
3112	`Internal error: List could not close the FORMSPEC formsfile.`	
3113	`List could not find form !in the FORMSPEC formsfile.`	
3114	`List write failed. (FSERR !!!!)`	
3115	`FORMSPECLINESPERPAGE JCW must be between 1 and 150. List failed.`	
3116	`FORMSPECLINESPERPAGE JCW must be between 1 and 150.`	

NOTE In most cases, the corrective action is clearly implied by the message. "Internal Errors" are serious; they imply a corrupted forms file and/or a program error. Call your SE.

Application-Ready Buffer (ARB) Errors

Error	Message	Cause/Action
3403	`Internal error: ARB detail record missing.`	
3405	`This field must not be blank.`	Enter the required information in the field.
3406	`This field accepts only contiguous digits.`	Remove any spaces between the digits.
3407	`Must specify number of decimals here.`	
3409	`Internal error: could not update ARB Detail record.`	
3412	`Internal error: Field Symbol Table record missing.`	
3413	`The form has no fields.`	You cannot reproduce the form fields on the ARB because they do not exist.
3414	`Internal error: Data Type Conversion record missing.`	
3416	`Internal error: Could not create ARB Detail record.`	
3418	`Internal error: could not create ARB Map record.`	
3420	`An ARB already exists.`	You cannot create two ARBs for one form. If you want to create a new ARB, you must delete the old one first.
3423	`Warning: Run-time errors may result from default data conversions.`	You may have specified incompatible data conversions. Check your data type to make sure that data conversions are legitimate. If your code has made provision for these anomalies, you may disregard the warning.
3424	`Warning: ARB length adjusted to match data type.`	You specified a length for an ARB field that was too long or too short.
3425	`Length must be greater than zero.`	
3426	`Field was not found in ARB.`	
3429	`ARB does not exist.`	
3433	`Form ! has duplicate field names.`	Correct the name duplication in the field menu and retry.
3450	`First field of range is missing; specify a name or number.`	

Error	Message	Cause/Action
3451	`Number specified is not in current ARB range.`	
3452	`ARB is full; no more fields may be added.`	An ARB may hold up to 256 fields.
3453	`Please enter a valid ARB field name.`	Check the syntax of the name you have entered.
3455	`Range is reversed; try 'leading-field/trailing field'.`	You have reversed the order of the fields you want to manipulate. The field that appears first on the ARB must appear first in the command.
3456	`Please enter a valid command (Add, Move, Rename or DELETE).`	
3457	`That command cannot be abbreviated. Please try again.`	
3458	`ARB field name is too long; maximum length is 15 characters.`	
3459	`Please specify Before or After, or leave blank.`	
3460	`ARB field name to be added already exists in ARB.`	You cannot duplicate an ARB field name; choose a new name or delete the other field first.
3461	`Internal error: Selection routine approved an unknown command.`	
3462	`New (Destination) name already exists in ARB.`	Select a name that does not already exist.
3465	`A destination entry requires that Before or After be specified.`	If you do not want to specify Before or After, you must leave the Destination field blank.
3466	`An ARB field name must start with an alphabetic character.`	
3467	`No imbedded blanks are allowed in an ARB field name.`	
3468	`Legal ARB field name characters are A-Z, 0-9 and '_'.`	
3469	`An ARB field name cannot ne a reserved word.`	Choose a new name for the field.
3473	`ARB field name must be specified; numeric reference is not allowed.`	

Error	Message	Cause/Action
3474	`Second field of range is missing; specify ARB field name or number.`	
3475	`Internal error: SearchRec did not locate field in ARB Add.`	
3476	`A range cannot be moved to a destination within itself.`	
3498	`Internal error: VGETBUFFER failed (COM'STATUS !!!!)`	
3499	`Internal error: VPUTBUFFER failed (COM'STATUS !!!!)`	

NOTE In most cases, the corrective action is clearly implied by the message. “Internal Errors” are serious; they imply a corrupted forms file and/or a program error. Call your SE.

ARB Compile Errors

Error	Message	Cause/Action
3500	`In ARB !, field ! has no corresponding screen field.`	Either delete the ARB field, or add a field to the associated screen.
3501	`In ARB !, field ! has an incorrect length.`	Change the ARB field length to match the length of the corresponding screen field.
3502	`In ARB !, field ! has an incorrect ARB type.`	Check the permitted conversion types and correct the ARB data type.
3503	`Internal error: IN ARB !,missing source record for field !.`	

More Menu Init and Processing Errors

Errors	Message	Cause/Action
7001	`Correct syntax is: "Field formname fieldtag/name", then menu specs.`	
7002	`Unable to open screen file.`	
7003	`WARNING: No new screenfile specified. Old screenfile was closed.`	
7004	`Correct syntax is: "ADD formname [\comments]".`	
7005	`Correct syntax is: "READD formname [new formname] [\comments]"`	
7006	`A screen layout file is not currently open.`	
7007	`End of file when reading screen layout file.`	
7008	`File system error when reading screen layout file.`	
7009	`Form name in screen file record not equal to current form name.`	
7010	`Cannot find screen record for form.`	
7011	`Cannot find screen record for form.`	
7012	`Write of screen record to screen layout file failed.`	

Errors	Message	Cause/Action
7013	`WARNING: if ADD command, form created with no screen.`	
7014	`WARNING: if READD, screen not updated (form renamed?)`	
7015	`Correct syntax is: "FKLABELS formname [¸eystring1...¸eystringn]".`	
7016	`Screenfile contains an invalid character.`	
7017	`Missing datatype parameter`	
7018	`Missing field type parameter.`	
7019	`Missing field enhancement parameter.`	
7020	`SELECTTERM update parameters must be separated by commas.`	
7021	`Missing new FORMSPEC language ID.`	
7022	`FIELD update parameters must be separated by commas.`	
7023	`Renumbered !.`	
7024	`Renumber was not necessary for !.`	
7025	`Form is family child; renumber family by specifying parent form !.`	
7026	`Internal error: unable to fetch form head.`	
7027	`There are no forms in the current file.`	
7028	`Creating new forms file.`	
7029	`Recovering file -- PLEASE WAIT.`	
7030	`Compiling form !`	
7031	`Number of errors = !!!! Number of warnings = !!!!`	
7032	`There are no compiled forms in the current file.`	
7033	`No update values specified. Command ignored.`	

Errors	Message	Cause/Action
7034	`**** End of compiled forms listing ****`	
7035	`Missing color parameter.`	
7036	`Missing 264x parameter.`	
7037	`Missing 307x parameter.`	
7038	`New function key labels must be preceded by` `	
7039	`End of command file.`	
7040	Forms file is being converted to new format.	
7041	`Missing ARB name`	
7042	Sorry. That command has not been implemented.	
7043	Missing ARB field name.	
7044	`Correct syntax is:`	
7045	`ADDARBFIELD ARBname fieldname [BEFORE/AFTER fieldname]`	
7046	`Missing field name in "BEFORE/AFTER fieldname".`	
7047	MODARBFIELD ARBname fieldname { length, type }	
7048	MOVEARBFIELD ARBname field [field] [BEFORE/AFTER FIELD]	
7049	`SCREENTOARB [ char, date, dig, num, imp ]`	
7050	`ARBTOSCREEN [ char, int, real, pack, zone`	
7051	`RENAMEARBFIELD ARBname oldfieldname newfieldname`	

Native Language Support Errors

Error	Message	Cause/Action
8000	`This edit statement is not valid for international forms files.`	

Error	Message	Cause/Action
8001	Edit contains data inconsistent with language of this forms file.	
8002	Edit contains language-dependent data.	
8003	Edit contains data not compatible with the language of this forms file.	
8004	Invalid initial value for International forms file.	
9001	Native language support software not installed.	
9002	Language specified is not configured on this system.	
9003	Character set specified is not configured on this system.	
9004	Internal error: National table is not present.	
9005	Internal error: Bad NLT extra data segment.	
9006	Internal error: Bad LDST extra data segment.	
9007	Error calling native language support intrinsic.	
9008	Error calling native language support intrinsic.	
9009	Error calling native language support intrinsic.	
9010	Internal error: NLINFO item out of range.	
9011	WARNING: Language not configured, change or hit ENTER to proceed.	
9012	This edit requires a configured language.	
9014	Attempted setting a language-dependent forms file to another language.	
9015	NATIVE-3000 is currently the only selection available.	
9016	Invalid date value for international forms file.	

Error	Message	Cause/Action
9017	`Internal Error: Error updating language ID - try again.`	
9070	`Internal Error: writing terminal selection source record failed`	
9500	`Language of forms file is not configured on this system.`	
9998	`Language ID must be 0 to 999 or -1 for international forms file.`	
9999	`Call to native language support intrinsic failed.`	

VPLUS Intrinsic Error Messages

VOPENTERM and VCLOSETERM Messages

Error	Message	Cause/Action
001	`Internal error: Terminal file initialization failed. failed.`	`FCONTROL` or `FSETMODE` intrinsic
002	`Your terminal is not supported by VPLUS.`	VPLUS works on the HP terminals listed in Appendix G.
003	`VPLUS does not run on a series I.`	VPLUS does not run on an HP Series I computer system.
004	`The terminal status request failed.`	`VOPENTERM's` request for terminal status failed. Check that strapping is correct. May also be data comm problem.
005	`The terminal allocation failed.`	Message used only with the 2640A terminal, which is not supported by VPLUS. Reserved words in the COMAREA are not initialized to zeroes.
006	`The terminal needs the H strap in (closed).`	Refer to Appendix G for instructions on how to strap your terminal for VPLUS.
007	`Internal error: HP2640 mode could not be correctly set.`	Possible block mode or data comm problem.
008	`Internal error: HP2645 mode could not be correctly set.`	See error 007.
009	`The terminal open failed. (FSERR !)`	**FOPEN** intrinsic failed.
010	`The terminal close failed. (FSERR !)`	**FCLOSE** intrinsic failed.
011	`Sorry, this device is not supported by VPLUS.`	Refer to Appendix G for a list of supported terminals.
012	`The HP2640B needs straps A, C, H in (closed); D, F, G out.`	You must strap this terminal (and the 2644A) yourself; other terminals are strapped automatically. (Refer to Appendix G.)
013	`The HP2644A needs straps A, C, H in (closed); D, G out.`	You must strap this terminal (and the 2640B) yourself; other terminals are strapped automatically. (Refer to Appendix G.)
014	`The HP2645A/41A/48A needs straps A, C, H, J in (closed); D, G out.`	Note that these terminals should be configured automatically by VPLUS.

Error	Message	Cause/Action
015	`Terminal type incorrectly determined.`	Possible block mode or data comm problem. Check straps. Note: 2645 multipoint straps should never be touched!
016	`Internal error HP2645A multipoint terminal mode not correctly set.`	See error 015.
017	`Comarea too short for run-time HP3075/6 terminals.`	The Comarea must be 85 words for HP 3075/6 terminals. Set terminals. Comarealen to 85.
018	`Non-page/block mode terminal types not supported.`	Redefine terminal type. (Note: VPLUS requires a terminal which supports page/block mode.)
019	`Internal error: FDEVICECONTROL should not return ccg.`	See System Manager.
021	`Invalid terminal type.`	TERMTYPE used is a printer terminal type

General Messages

Error	Message	Cause/Action
020	`Error detected by host language.`	Check the Comarea item FILERRNUM for the host language error number.
021	`Invalid terminal type.`	See 021 above.
023	`Environment control file access failed.`	

VOPENFORMF and VCLOSEFORMF Messages

Error	Message	Cause/Action
040	`Internal error: DLSIZE failed; there is not enough DL area space.`	Stack size or MAXDATA too small. Re-PREP your application program with a larger MAXDATA parameter.
041	`Internal error: DLSIZE failed; the stack is frozen.`	Stack size or MAXDATA too small. Re-PREP your application program.
042	`Formsfile name not passed to VOPENFORMF.`	The name of the forms file to be opened must be included in the call to `VOPENFORMF`.
043	`Comarea length may not be less than 60.`	The Comarea length as specified by the item Comarealen must be 60 words for HP terminals and 85 words for data capture devices.
044	`Comarea length is too large.`	The Comarea length as specified by the item comarealeng must not be longer than the longest length documented.
045	`TERMINAL or FORMSFILE has not yet been opened.`	
046	`Unrecognized Comarea language code passed.`	
050	`Forms File open failed. (FSERR !)`	`FOPEN` intrinsic failed. Check forms file name.
051	`The file is not a VPLUS Forms File.`	Check the formfile parameter to make sure the file name is correct.
052	`Internal error: Forms File FGETINFO failed. (FSERR !)`	`FGETINFO` intrinsic failed.
053	`Forms File probably hasn’t been compiled. (FSERR !)`	Run FORMSPEC and compile the forms file.
060	`The program supplied COMAREA extension is too small.`	If you are using BASIC, check word 4 of COMAREA (USRBUFLEN), and increase the length of the COMAREA extension.

Error	Message	Cause/Action
061	`Internal error: Failure to obtain required PASCAL Heap area.`	
062	`Internal error: Failure to return PASCAL Heap area.`	
063	`Failure to close Forms File. (FSERR !)`	An error occurred while trying to close the forms file. Check that the forms file is open before VCLOSEFORMF is called.
067	`Invalid form store size specified.`	A number outside the range of -1 to 255 is specified in the FORM’STORE’SIZE variable, which is the 39th word of the VPLUS communication area.
068	`VPLUS internal error; there is not enough DL area space.`	For languages other than BASIC,
069	`VPLUS internal error; the stack is frozen.`	See error 041.
9001	`Native Language Support software not installed`	Check with System Manager to install Native Language Support software.
9002	`Language specified is not configured on this system.`	Select another language or check with System Manager to configure the desired language.
9500	`Language of forms file is not configured on this system.`	See System Manager to configure the language or use forms files on a system with that language configured.

VOPENBATCH and VCLOSEBATCH Messages

Error	Message	Cause/Action
070	`Forms file was re-complied since last using this batch.`	You may want to specify a different batch file name since the forms file has been changed since the specified batch file was last used with the forms file.
071	`A forms file must be opened before opening a batch file.`	Call `VOPENFORMF` before calling `VOPENBATCH`
072	`The file does not appear to be an existing batch file.`	Check the batch file name you specified; if this is a new file, it will be created automatically and then opened.
073	`A different forms file created this batch.`	You opened a forms file that is not the forms file associated with the batch files specified. See 070.
074	`Batch file not yet opened.`	
080	`Batch file open failed (FSERR !)`	Refer to the file system error to determine the cause of the open failure. Check file name.
081	`Can't read the batch file's user label (FSERR !)`	`FREADLABEL` intrinsic failed.
082	`Internal error: Batch file FGETINFO failed. (FSERR !)`	`FGETINFO` intrinsic failed.
084	`Can't write user label to batch file (FSERR !)`	`FWRITELABEL` intrinsic failed.
085	`Internal error: Batch file close failed. (FSERR !)`	`FCLOSEBATCH` intrinsic failed. Refer to the file system error to determine the cause of the close failure.

Access Method Messages

Error	Message	Cause/Action
086	`Access method: file code error.`	
087	`Access method: Attempt to add a duplicate key.`	
088	`Access method: Internal error. Key insertion location error.`	
089	`Access method: Internal error. Block not present.`	
090	`Access method: Attempt to open an old file as new.`	
091	`Access method: Internal error. Key block buffer error.`	

Error	Message	Cause/Action
092	`Access method: Internal error. Invalid key block number.`	
093	`Access method: Internal error. Block not verified.`	
094	`Access method: Record not found.`	
095	`Access method: Not enough space in extension for the directory.`	Fast form files require 500 words of space in the COMAREA extension. If you are coding in BASIC, list the forms file to find how many words are needed for the extension, and increase the COMAREA field USERBUFLEN to this number. If the application is not written in BASIC, increase the MAXDATA parameter on the PREP or RUN command.
096	`Access method: Internal error. Parent block not found.`	
097	`Access method: Internal error. Illegal entry number.`	
098	`Access method: The file is not a KSAMless forms/ref file.`	
099	`Access method: The file is at EOF.`	

VGETNEXTFORM Messages

Error	Message	Cause/Action
100	`Can't find the next form. (FSERR !) program.`	Check form name in application.
101	`Form $HEAD does not apply, since no forms have been displayed.`	NFNAME in COMAREA is $HEAD, but this specification only applies when forms have been displayed.
102	`An invalid COM'REPEAT value was supplied programmatically.`	REPEATAPP in COMAREA should be set to one of the codes: 0 - no repeat/append 1 - repeat current form 2 - repeat & append current form
103	`An invalid COM'NFOPT value was supplied programmatically.`	The next form option in FREEZAPP of COMAREA can be: 0 - clear current form, display next form 1 - append next form to current form 2 - freeze current form, append next form

Error	Message	Cause/Action
104	`Formsfile not compiled for this run-time terminal.`	The type of terminal is being used has not been selected for use with the forms file. Specify the terminal type in the FORMSPEC Terminal Selection menu, then recompile the forms file. If you open a forms file and then call `VGETNEXTFORM` without opening the terminal, you must compile the forms file for a 264X terminal
105	`HP3075/6 terminal with numeric only display is not supported.`	VPLUS does not run on the numeric display HP3075/6 terminals.
106	`HP3077 terminal is not supported except for opening and closing.`	VPLUS only opens and closes the HP3077 terminal; no other functions can be performed.

VMERGE Messages

Following are the messages produced by the VMERGE utility. The cause of each, and the action you may take is also described.

Message	Cause/Action	Action
`hp32209V.UU.FF VMERGE (c) Hewlett-Packard Co. 1992 ALL RIGHTS RESERVED`	Header generated at the start of VMERGE processing.	No action required.
*****Internal processing error (VM1001)	An unexpected condition has been detected by VMERGE.	Preserve the input files for diagnosis and contact HP rep.
`Begin VMERGE Processing ... (VM1003)`	Given at the start of processing.	No action required.
`No consistency check required, AUX file is $NULL`	Self-explanatory.	
`Consistency check of MASTER and AUX file begun ...(VM1005)`	Self-explanatory.	
`Input files are consistent (VM1006)`	The consistency check of MASTER and AUX has been completed, and the files have been found consistent.	No action required.
`***** Input files are not consistent (VM1007)`	The consistency check of MASTER and AUX has been completed, and the files have been found inconsistent. (Normally an explanatory message indicating why they are inconsistent will have been issued prior to this one.) The OUTPUT file will not be produced.	Adjust the input files to overcome the indicated inconsistency and rerun VMERGE
`End of VMERGE Processing. (VM1009)`	Self-explanatory.	None
`***** Form name table could not be allocated (VM1400)`	An attempt to allocate an extra data segment to hold a form name table for the MASTER or AUX forms has failed.	Consult with your system administrator to determine why this failure occurred on your system. This is probably due to limitations on the number of Extra Data Segments that can be created in your configured environment

Message	Cause/Action	Action
`***** Form name table overflow (VM1401)`	More forms have been found in the MASTER or AUX file than can be accommodated in the form name table.	Contact your HP rep.
`***** Conflicting data in MASTER forms file (VM1402)`	Invalid or inconsistent data has been found in the MASTER file.	Recreate the file and rerun VMERGE. If the problem persists, contact your HP rep.
`***** Conflicting data in AUX forms file (VM1403)`	Invalid or inconsistent data has been found in the AUX file.	Recreate the file and rerun VMERGE. If the problem persists, contact your HP rep
`**** Error accessing table in an extra data segment (VM1404)`	An attempt to access data in an extra data segment was rejected by the MPE operating system.	Rerun VMERGE. If the problem persists, contact your HP rep.
`***** Version of MASTER file format not supported by VMERGE (VM1500)`	The MASTER file was compiled with a version of FORMSPEC that is not supported by VMERGE. (Versions prior to B.03.03 are not supported.)	Reconstruct the file with the current version of FORMSPEC.
`***** Version of AUX file format not supported by VMERGE (VM1501)`	The AUX file was compiled with a version of FORMSPEC that is not supported by VMERGE. (Versions prior to B.03.03 are not supported.)	Reconstruct the file with the current version of FORMSPEC.
`Warning: Error enhancement settings differ (VM1504)`	The error enhancement settings differ in the MASTER and AUX files.	None. (The value from the MASTER file is retained in OUTPUT.
`Warning: Window display line settings differ (VM1505)`	The display line settings differ in the MASTER and AUX files.	None. (The value from the MASTER file is retained in OUTPUT.)
`Warning: Error window color settings differ (VM1506)`	The error window color settings differ in the MASTER and AUX files.	None. (The value from the MASTER file is retained in OUTPUT.)
`Warning: Window enhancement settings differ (VM1507)`	The window enhancement settings differ in the MASTER and AUX files.	None. (The value from the MASTER file is retained in OUTPUT.
`***** Terminal Specific settings differ (VM1508)`	The terminal settings differ in the MASTER and AUX files.	Revise the files to make them compatible and rerun the utility
`***** Language ID settings differ (VM1509)`	The language id settings differ in the MASTER and AUX files.	Revise the files to make them compatible and rerun the utility.

Message	Cause/Action	Action
`***** Number of databases differ (VM1510)`	The number of databases differ in the MASTER and AUX files.	Revise the files to make them compatible and rerun the utility
`***** Save fields specifications differ (VM1521)`	The save field specifications differ in the MASTER and AUX files.	Revise the files to make them compatible and rerun the utility. The save field specifications must be identical in all respects in both input files.
`***** Global function key labels differ (VM1531)`	The global function key labels differ in the MASTER and AUX files.	Revise the files to make them compatible and rerun the utility
`***** 307x terminal settings differ (VM1541)`	The 307x terminal settings differ in the MASTER and AUX files.	Revise the files to make them compatible and rerun the utility
`Form name "nnnnnnnnnnnnnnn" found in MASTER file (VM1601)`	Documents that the named form was found in the MASTER file	None
`Form name "nnnnnnnnnnnnnnn" found in AUX file (VM1603)`	Documents that the named form was found in the AUX file.	None
`***** MASTER and AUX contain duplicate form name(s) (VM1611)`	Forms with the same name were found in both the MASTER and AUX files	Remove or rename the duplicate forms from one file and rerun VMERGE.
`***** Duplicate form name: nnnnnnnnnnnnnnn (VM1612)`	Identifies a form name found in both the MASTER and AUX files.	Remove or rename the duplicate form one file and rerun VMERGE
`***** Unattached messages; use fast forms files (VM1613)`	Messages have been found that are not associated with any form. OUTPUT file is not produced.	Compile the input files and produce fast forms files. Use the fast forms files as input to VMERGE
`***** MASTER file could not be opened (VM1700)`	The file designated as MASTER could not be opened.	Remove the impediment to opening the file and rerun VMERGE. (The file may not exist under the designated name, or may be already open for exclusive use, etc.)

Message	Cause/Action	Action
`***** Error reading MASTER file (VM1701)`	An error occurred in attempting to read the MASTER file.	Regenerate the file and rerun VMERGE.
`***** Error in MASTER file data (VM1702)`	An inconsistency was found in the data in the MASTER file.	Recreate the file and rerun the utility. If the problem persists contact your HP rep.
`MASTER file opened (VM1703)`	Self-explanatory.	None
`***** MASTER file type is not VFORM or VFAST (VM1704)`	Self-explanatory	Designate the correct file and rerun VMERGE.
`MASTER file designated as: fff ... f (VM1705)`	Shows the full pathname of the MASTER file.	None
`***** MASTER file has not been compiled (VM1706)`	Self-explanatory.	Compile the MASTER file and rerun VMERGE.
`***** AUX file could not be opened (VM1800)`	The file designated as AUX could not be opened.	Remove the impediment to opening the file and rerun VMERGE. (The file may not exist under the designated name or may be already open for exclusive use, etc.
`***** Error reading AUX file (VM1801)`	An error occurred in attempting to read the AUX file.	Regenerate the file and rerun VMERGE.
`***** Error in AUX file data (VM1802)`	An inconsistency was found in the data in the AUX file.	Recreate the file and rerun the utility. If the problem persists contact your HP rep.
`AUX file opened (VM1803)`	Self-explanatory.	None
`***** AUX file type is not VFORM or VFAST (VM1804)`	Self-explanatory.	Designate the correct file and rerun VMERGE.
`AUX file designated as: fff ... f (VM1805)`	Shows the full pathname of the AUX file.	None.
`***** AUX file has not been compiled (VM1806)`	Self-explanatory.	Compile the AUX file and rerun VMERGE.
`***** OUTPUT file could not be opened (VM1900)`	Self-explanatory.	Remove the impediment to opening the OUTPUT file and rerun VMERGE.

Message	**Cause/Action**	**Action**
`**** Error writing OUTPUT file (VM1901)`	Self-explanatory.	Correct the problem that has caused the write error and rerun VMERGE. You may need to consult with your system administrator to determine the problem.
`OUTPUT file opened (VM1903)`	Self-explanatory.	None.
`***** OUTPUT file type is not VFAST (VM1904)`	The file designated as OUTPUT pre-exists but is not of type VFAST.	This is viewed as an error since it is assumed that the user has unintentionally designated the name of a file being used for other purposes. Redesignate the OUTPUT file (or purge the old file if the name is correct) and rerun VMERGE.
`OUTPUT file designated as: fff ... f (VM1905)`	Shows the full pathname of the OUTPUT file	None
Error closing OUTPUT file (VM1906	Self-explanatory.	Correct the problem that has caused the error and rerun VMERGE. You may need to consult with your system administrator to determine the problem
`***** OUTPUT file purged (VM1911)`	Inconsistency or some other problem has prevented the OUTPUT file from being successfully generated. To avoid confusion, the file is purged.	Correct the underlying problem and rerun VMERGE
`***** OUTPUT file pre-existed and could not be purged (VM1912)`	An attempt to purge the existing file with the name designated for the OUTPUT file failed.	Make sure that the correct file name has been designated. If so, determine why it could not be purged and remove the impediment

VSHOWFORM Messages

Error	Message	Cause/Action
130	`Internal Error: Terminal write failed. (FSERR !)`	FWRITE intrinsic failed.
131	`Internal error: Color conversion failed.`	

VREADFIELDS Messages

Error	Message	Cause/Action
150	`The supplied read buffer (dbuf) is too small.`	The data buffer in memory is too small for the data on the form. Check application program DBUF array size and DBUFLEN.
151	`Internal error: An expected DC2 from the terminal was missing.`	Block mode or data comm problem. Press **REFRESH**.
152	`Internal error: A read terminator (an RS or a GS) was expected.`	Press ENTER again. Also try **REFRESH**. If problem recurs, check DBUFLEN in application program. Possible block mode or data comm problem.
153	`A status request was pending, so the terminal read was invalid.`	Application program requested terminal status, then called VREADFIELDS.
154	`An unrecognized escape sequence was read.`	Possible softkey problem. Press **REFRESH** or perhaps **RESET** to correct.
155	`Read length error.`	Length of data actually read is less than expected for current form.
160	`Internal error: Terminal Read Failed (FSERR!)`	FREAD intrinsic failed. If user timeouts were enabled in the user application, a time out has occurred.
161	`Field too long for HP3075/6 terminals.`	The maximum length of a field on the HP 3075/6 terminals is 200 characters.
162	`Terminal Power Failure detected.`	A power failure has been detected on an HP 3075/6 terminal.
163	`Invalid error light.`	Valid error lights are letters of the alphabet or the @@ character.
164	`HP3075/6 Terminal printer is out of paper.`	Load the printer with a roll of paper.
165	`Device is not supported by VPRINTLOCAL.`	The terminal must be an HP 3075A or HP 3076A.
166	`HP3075/6 Terminal does not have a printer.`	Your terminal is not equipped with a printer.

VPRINTFORM Messages

Error	Message	Cause/Action
190	`Can't open the printfile. (FSERR !)`	`FOPEN` intrinsic failed. Refer to file system error to determine the cause.
191	`Can't write to the print file. (FSERR !)`	`FWRITE` intrinsic failed.

VINITFORM, VFIELDEDITS, and VFINISH Messages

Error	Message	Cause/Action
202	`The required field is empty.`	A required field in current form is blank.
203	`Invalid field type.`	Valid field types are: 0(optional), R(required), D(display only), P(Process even if blank).
204	`The field can only contain digits.`	The field is specified as type DIG, but it contains characters other than 0-9.
205	`The field can only contain a number.`	The field is specified as type NUM, but it contains characters other than 0-9, decimal point, comma, or plus or minus sign.
206	`The field can only contain a number, with max ! decimal places.`	Reduce the number of digits to the right of the decimal point with max ! decimal to n.
207	`The field can only contain a number (! decimal places implied).`	The field is specified as type IMPn, but it contains characters other than 0-9, implied). decimal point, comma, or plus or minus sign. Also, there could be more than n digits to the right of the decimal point.
208	`The field must be a valid date, in DMY order.`	The data in a DMY field is not a valid date, or is not in order: day, month, year.
209	`The field must be a valid date, in MDY order.`	The data in a DMY field is not a valid date, or is not in order: day, month, year.
210	`The field must be a valid date, in YMD order.`	The data in a YMD field is not a valid date, or is not in order: year, month, day.
211	`Internal error: Invalid field data type.`	Field table problem.
212	`Error during data conversion between family members.`	A data conversion error has been detected by `VINITFORM` when converting between family members. Check for field type compatibility problems in the form family in the forms file.
220	`Internal error: Invalid destination.`	Code execution problem.

Error	Message	Cause/Action
222	`An edit test failed.`	The default field edit error message had been provided. Check the data you entered in the field against the processing specifications.
321	`Attempted a SET into a numeric field that is too short.`	The destination of a SET statement has fewer characters too short. than the number being moved to it.
323	`Internal error: Invalid op code.`	Code execution problem.
325	`Attempted a SET into a date field that is too short.`	The destination of a SET statement has fewer characters than the value being moved to it.
326	`Internal error in field.`	
327	`User-defined error for this field.`	A FORMSPEC custom error message is associated with this field, but could not be correctly retrieved from forms file.
328	`The field cannot contain a negative number (Dtype DIG).`	A minus sign is in a field whose data type is DIG, only digits (0-9) allowed.
330	`Internal error: Invalid op in logical eval.`	Code execution problem.
331	`Range error. Low value is greater than high value.`	The first value is in an IN/NIN range is greater than the second value (for a:b, a>b)
332	`Check digit requested on empty field.`	CDIGIT is specified for a field that contains no data. There must be a value in order to test the check digit.
333	`Check digit requested on field containing only + or - sign.`	CDIGIT is specified for a field that has no value except a sign; there must be a value in order to test the check digit.
334	`Check digit requested on field containing special characters.`	CDIGIT only operates on numeric (0-9) or alphabetic (A-Z) values. characters.
335	`Check digit 10 is invalid for modulus 11 calculation.`	Since value has remainder greater than 9, it has no possible modulus 10 check digit. Use CDIGIT 11.
336	`Internal error: Check digit modulus must be 10 or 11.`	
337	`Check digit validation string contains only one character.`	A field whose value is to be checked by a modulus calculation must contain at least two characters.
340	`Division by zero was attempted.`	Expression in field processing statement evaluated to division by zero.

Error	Message	Cause/Action
341	`Attempted a SET from a field containing an invalid number.`	A numeric value is expected as the source of a SET statement, but the source is not a valid number.
344	`Internal error: Invalid op in long eval.`	
345	`Overflow in an add operation.`	Value is outside 4-word REAL range. Range of 4-word REAL is from approximately 8.636 times 10^(-78) to 1.158 times 10^77.
346	`Overflow in a subtract operation.`	See error 345.
347	`Overflow in a multiply operation.`	See error 345.
348	`Overflow in a divide operation.`	See error 345.
349	`Overflow in a percent operation.`	See error 345.
350	`Index expression out of range in an "index of" numeric statement.`	In an index retrieve operand (index OF element, element...) the index evaluates to a digit greater than the number of elements in the list. (Numeric operands).
351	`Overflow in a negate operation.`	See error 345.
360	`Attempted a SET from a field containing an invalid date.`	A date is expected as the source of a SET statement, but the date is not valid.
363	`Internal error: Invalid op in double eval.`	
364	`Index expression out of range in an "index of" date statement.`	See error 350. (Date operands)
370	`Internal error: Invalid op in text eval.`	
371	`Index expression out of range in an "index of" text statement.`	See error 350. (CHAR operands)
380	`The index expression evaluates to "empty."`	An arithmetic expression used as the index in an index OF element, element.. operand is blank or zero.

VREADBATCH Messages

Error	Message	Cause/Action
400	`Can't read the batch file record. (FSERR !)`	`FREADDIR` intrinsic failed.
401	`Warning: Can't browse a batch file with variable length records.`	Browse mode is illegal for batch records which are not fixed length.

VWRITEBATCH Messages

Error	Message	Cause/Action
450	`Can't write the batch record. (FSERR !)`	`FWRITEDIR` intrinsic failed.

VSETERROR, VGETFIELD, VPUTFIELD, VPUTtype, and VGETtype Messages

Error	Message	Cause/Action
500	`A field with the field number supplied does not exist.`	The fieldnum parameter contains a value that is not a field number in the current form.
501	`The field number supplied is out of range.`	The fieldnum parameter contains a value greater than any field number associated with the current form.
502	`The field requested is in error.`	The field specified by the fieldnum parameter contains invalid data.
503	`The field requested is empty.`	The field specified by the fieldnum parameter contains no data.
504	`Error converting field to numeric type.`	`VGETtype` tried to convert value to a numeric type, and failed. Check conversion rules. (For `VGETINT`, message, indicates attempt to convert number > 32767; receiving field is unchanged.)
505	`Error converting numeric type to ASCII.`	`VPUTtype` tried to convert a numeric value to a character string and failed. Check conversion rules.
506	`A DIGIT field cannot contain a negative number.`	A value in a DIG type field has a minus sign; DIG values may only be digits (0-9).
507	`Function key labels option is disabled.`	The COMAREA item LABEL'OPTION must be set to one before opening the forms file.

Error	Message	Cause/Action
508	`Error reading function key values from forms file (FSERR !!!!)`	An error occurred while trying to read the function key label definitions from the forms file.
509	`Invalid input parameter to function key intrinsics.`	An invalid value has been passed as the FORM-OR-GLOB or KEY-NUM parameter to VGETKEYLABELS, VSETKEYLABEL, or VSETKEYLABELS intrinsics.
510	`A field with the screen order supplied does not exist.`	
515	`WARNING: Sign in field not stored in unsigned variable.`	If the NUMDIGITS is positive, the intrinsic assumes the destination variable for the GET is unsigned. If the parameter is negative, the destination is assumed to be signed.

VGETBUFFER and VPUTBUFFER ARB Messages

Error	Message	Cause/Action
519	`ARB flag is set but no ARB record is found.`	This is an application-specified conversion, but the forms file does not contain an ARB record. Do the following: 1. Initialize ARB conversion flag to 0 if conver sion is not desired. 2. Use FORMSPEC to add the necessary ARB record. 3. Check for possible corruption of the formsfile.
520	`ARB conversion type specified is not supported.`	Check for possible corruption of the formsfile.
521	`ARB record length exceeded maximum limit.`	Check for possible corruption of the formsfile.
522	`IEEE conversion is not supported in this version.`	This application-specified IEEE conversion is not yet supported.Check the IEEE conversion flag in the Comarea.
523	`ARB record does not contain the minimum amount of informati on required.`	Check for possible corruption of the formsfile.
524	`ARB record does not contain the minimum amount of information required.`	
525	Error converting YMD type to YYMMDD format.	
526	`Error converting MDY type to YY MMDD type.`	
527	`Error converting DMY type to YY MMDD format.`	
528	`Internal error: converting Julian to YYMMDD format.`	
529	`YYMMDD variable parameter does not contain a valid date.`	
530	`Internal error: converting Julian to date type.`	
531	`Attempted to place date  into s hort date field (less than 8 char).`	

VGETLANG and VSETLANG Messages

Error	Message	Cause/Action
9002	`Language specified is not configured on this system.`	Select another language or check with System Manager to configure the desired language.
9014	`Attempted setting a language dependent forms file to another language.`	`VSETLANG` can only be used with international forms files.

VCHANGEFIELD Messages

Error	Message	Cause/Action
850	`Invalid field type specification.`	Field type incorrect; use one of the listed codes.
851	`Invalid data type specification.`	Data type incorrect; use one of the listed codes - n indicates number of decimal positions.
852	`Invalid enhance type specification.`	Enhancement code is incorrect; check and correct.
853	`Internal error: invalid field type.`	Internal.
854	`Internal error: invalid data type.`	Internal.
855	`Internal error: invalid enhance type.`	Internal.
856	`Invalid VCHANGEFIELD specification type.`	Value must be 1-6.

VPLACESURSOR Intrinsic

Error	Message	Cause/Action
800	`Cannot place cursor to display only fields.`	Field number must be for an input field.

INFO Intrinsic Messages

Error	Message	Cause/Action
601	`Invalid input parameters to INFO intrinsic.`	
602	`Form specified to FORMINFO does not exist.`	
603	`INFO intrinsic: Form field list read error (FSERR !!!!).`	

Error	Message	Cause/Action
604	`INFO intrinsic: Form record read error (FSERR !!!!).`	
605	`FF version is old. Must recompile to use FORMINFO intrinsic.`	
606	`Invalid field specified to FIELDINFO intrinsic.`	Make sure that the field name is correctly spelled if the field name is the key; make sure that the field name parameter is 16 bytes long, and padded with ASCII blanks. If the field number is used as the key, be certain that this number is the actual field number.
607	`Invalid form # specified to FORMINFO intrinsic.`	

Forms Loading Messages

Error	Message	Cause/Action
700	`No such form in terminal form storage memory.`	The specified form is not purged because it does not exist in the form storage directory.
701	`Terminal form storage memory is full.`	The specified form is not loaded because the form storage memory is full.
702	`Requested loan of blankform name.`	

SNA DHCF Intrinsic Messages

Native Language Support

Error	Message	Cause/Action
8000	`This edit statement is not valid for international forms files.`	
8001	`Edit contains data inconsistent with language of this forms file.`	
8002	`Edit contains language dependent data.`	
8003	`Edit contains data not compatible with the language of this forms file.`	
8004	`Invalid initial value for international forms file.`	
9001	`Native Language Support Software not installed.`	Check with System manager to install NLS software.
9002	`Language specified is not configured on this system.`	Select another language or check with System Manager to configure the desired language.
9003	`Character set specified is not configured on this system.`	
9004	`Internal error: National table is not present.`	Internal error from NLS.
9005	`Internal error: Bad NLT extra data segment.`	Internal error from NLS.
9006	`Internal error: Bad LDST extra data segment.`	Internal error from NLS.
9007	`Error calling native language support intrinsic`	Internal error from NLS.
9008	`Error calling native language support intrinsic.`	Internal error from NLS.
9009	`Error calling native language support intrinsic`	Internal error from NLS.
9010	`Internal error: NLINFO item out of range.`	Internal error from NLS.
9011	`WARNING: Language not configured, change or hit ENTER to proceed.`	Language specified is not configured on the system; forms file produced will only run on a system configured with that language.

Error	Message	Cause/Action
9012	`This edit requires a configured language.`	
9013	`Encountered invalid Asian (2-byte) character.`	
9014	`Attempted setting a language dependent forms file to another language.`	FORMSPEC language can only be 0 in this version.
9015	`NATIVE-3000 is currently the only selection available.`	FORMSPEC language can only be a 0 in this version.
9016	`Invalid date value for international forms file.`	
9017	`Internal error: Error updating language ID - try again.`	
9070	`Internal error: writing terminal selection source record failed.`	
9500	`Language of forms file is not configured on this system.`	
9998	`Language ID must be 0 to 999 or -1 for international forms file.`	Forms file language ID must be between -1 and 999.
9999	`Call to native language support intrinsic failed.`	

REFSPEC Messages

Access Method Errors

Error	Message	Cause/Action
086	`Access method: file code error.`	
087	`Access method: Attempt to add a duplicate key.`	
088	`Access method: Internal error. Key insertion location error.`	
089	`Access method: Internal error. Block not present.`	
090	`Access method: Attempt to open an old file as new.`	
091	`Access method: Internal error. Key block buffer error.`	
092	`Access method: Internal error. Invalid key block number.`	
093	`Access method: Internal error. Block not verified.`	
094	`Access method: Record not found.`	
095	`Access method: Not enough space in extension for the directory.`	
096	`Access method: Internal error. Parent block not found.`	
097	`Access method: Internal error. Illegal entry number.`	
098	`Access method: The file is not a KSAMless forms/ref file.`	
099	`Access method: The file is at EOF.`	

Output Sequence Definition Errors

Error	Message	Cause/Action
101	`Field name has more than 15 characters.`	Limit name of input field to 15 characters.
102	`Field name improperly aligned on screen.`	Enter input field name at tab position 1.
103	`Form name has more than 15 characters.`	Limit name of input form to 15 characters.
104	`Form name ! is not ininput forms sequence.`	A form name on the OUTPUT RECORD menu was not named on corresponding INPUT FORMS menu.
105	`A digit is not a legal character in this field.`	A field or form name is expected in this field.
107	`Substring start column offset is improperly aligned on screen.`	Enter input substring start column at tab position 2.
108	`Substring starting column has more than 4 digits.`	Substring start column must not exceed 4 digits.
111	`Closing quote expected.`	Character string constant must be enclosed within quotes; the closing quote is missing.
113	`Input length field is improperly aligned on screen.`	Enter input substring length at tab position 3.
114	`Internal error: Expected LF after CR.`	Line feed expected after carriage return.
115	`Line must start with field name, literal or system constant.`	First column of OUTPUT RECORD menu should contain first character of field name, a literal, or system constant.
116	`Input length cannot have more than 4 digits.`	Input substring length must not exceed 4 digits.
118	`Field name cannot be blank.`	A field name must be specified if other characteristics are entered on line.
119	`Letters are not allowed in this field.`	Only digits are expected in this field.
121	`Form name is improperly aligned on screen.`	Enter input form name at tab position 4.
123	`Output field name has more than 15 characters.`	A field name is limited to 15 characters.
124	`Output field name is improperly aligned on screen.`	Enter output field name at tab position 5.
126	`Length of output field may only have up to 4 digits.`	Output field length must not exceed 4 digits.

Error	Message	Cause/Action
127	`Output field length is improperly aligned on screen.`	Enter output field length at tab position 7.
129	`Output field starting offset can only have up to 5 digits.`	Starting column of output field must not exceed 5 digits.
130	`Output field starting offset is improperly5 digits.`	Starting column of output field must not exceed 5 digits.
131	`Start record indicator is not in its field.`	Enter start-of-record marker at tab position 8.
132	`Start record indicator should be a single character.`	A start-of-record marker is limited to 1 character.
133	`Column numbering starts with 1.`	The starting column in the output record cannot be zero.
134	`This is not a legal system constant.`	Use one of the legal constants: $CR, $LF, $GS, $US, $RS, or $ followed by the numeric equivalent of an ASCII character. Other constants may not start with a $.
135	`Output field length must be a positive integer greater than 0.`	Output length cannot be zero or a negative number.
136	`Output field starting column causes overlapping fields.`	The starting column for this field falls within the bounds of a previous field. Enter a higher value.
137	`A single literal cannot have more than 75 characters.`	Literal must fit on a single line of OUTPUT RECORD menu and must not overwrite the start-of-record marker position.
138	`Literal must not continue beyond column 75.`	One line of OUTPUT RECORD menu is limited to columns 1-75.
139	`This is not a field. No character may appear here.`	
140	`Only literal, ASCII constant or system constant expected here.`	
141	`Digit must follow sign.`	
142	`Value for ASCII character must be between 0 and 127.`	The numeric equivalent of an ASCII character may be only 0-127.
143	`Field name must start with char. No embedded blanks allowed.`	First character of field name must be alphabetic; field name may not have embedded blanks.
144	`Record size must have a value.`	Record size on globals menu is a required field. User must give maximum size of output record.

Error	Message	Cause/Action
145	`"Y" or "N" expected in upshift field.`	On GLOBALS menu, only a Y or N can be entered in response to "Upshift?"; default is N.
146	`"Y" or "N" expected in answer to "Convert to EBCDIC?".`	On GLOBALS menu, only a Y or N can be entered in response to "Convert to EBCDIC?"; default is N.
147	`Illegal character.`	Character in output record menu is neither alphabetic nor numeric.
148	`This is not a field.`	In output record menu, you may enter data only in the highlighted fields.
149	`Form name must start with char. No embedded blanks allowed.`	First character of form name must be alphabetic; name may not have embedded blanks.
150	`Output field name must start with char. No embedded blanks allowed.`	See error 143.
151	`Internal error: Field alignment check failed.`	
153	`Fields cannot have embedded blanks.`	Embedded blanks are not allowed in any field of the OUTPUT RECORD menu.
154	`Input length field cannot have embedded blanks.`	Length of input field must be a positive integer with no embedded blanks.
155	`Start substring field cannot have embedded blanks.`	Input starting column must not contain embedded blanks.
157	`Output starting column field cannot contain blanks.`	Embedded blanks are not allowed as part of the output starting specification.
158	`Input length field cannot contain blanks.`	No embedded blanks are allowed in the input length field.
159	`Input length must be greater than 0.`	Specify input length as positive integer greater than zero.
160	`Substring starting column exceeds length of forms file field.`	Start of input field must be within the forms file field; column specified is greater than the number of characters in the form field.
161	`Substring start column + input length > length of forms file field.`	Specified starting column plus length of field add up to more characters than the forms file field contains.
162	`Input length for reformat field exceeds length of forms file field.`	Length specified for input field is greater than the number of forms file field. characters in the field.
163	`Output record size must be between 1 and 8192.`	The output record size exceeds 8192 characters. Make the output record smaller.

Validation Errors

Error	Message	Cause/Action
201	`Field ! not found in form !.`	Input field in OUTPUT RECORD menu not found in specified form.
202	`Field ! not found in any form in input forms sequence.`	Input field in OUTPUT RECORD menu not found in any form in the sequence. corresponding INPUT FORMS menu.
203	`Reformat identifier ! is not a unique reformat sequence.`	First form specified on INPUT FORMS menu is not unique to the identifier. reformat file. Use a form name that has not been specified as a reformat identifier on another menu.
204	`Internal error: Data type is bad.`	See error 206.
205	`Output record length is greater than globally defined record length.`	Change either the total output record length on the OUTPUT RECORD defined record length. menu, or the record length on the GLOBALS menu, or start a new record using the start record marker.
206	`Internal error: Bad entry in field table. Illegal field type.`	Internal descriptions of fields are bad. Press ENTER again. If field type. that fails, delete and recreate the offending format.
207	`Error opening reformat file. (FSERR !!!!)`	Refer to the file system error for the cause of the open failure.
208	`A unique reformat identifier is required.`	See error 203.
210	`Internal error: OFST table is bad.`	See error 206. OFST is Old Field Symbol Table.
211	`Internal error: Field table is bad. Set'next failed.`	See error 206.
212	`Form name must begin with an alphabetic character.`	Form name cannot start with a special character or a digit.
213	`Form name cannot have embedded " "'s or "."'s.`	The only non-alphanumeric character allowed in a form name is the underline "_".
214	`Form name may contain only alphas, digits and "_"'s.`	Use the correct naming conventions: A name may contain letters, digits, or the underline, must start with a letter, and have no more than 15 characters.
215	`Internal error in ADJUST'ALL'START'COL.`	Internal error in recording the change in the default starting position of each field, a change resulting from altering the field or record separators in the globals menu.

Error	Message	Cause/Action
216	`Internal error reading reformat intermediate record for !.`	Error reading an internally used record from the reformat file.
217	`WARNING: Start column goes beyond global output record length.`	Starting column for output record specified on OUTPUT RECORD menu is greater than the record length on the GLOBALS menu. To avoid truncation, change one of the two specifications.
218	`Embedded blank form names not allowed in a sequence.`	A form name consisting of a blank found in the INPUT FORMS menu. Remove the blank line from the sequence.
219	`Internal error: Data type is bad.`	See error 204.
220	`WARNING: This field causes output record size to be too big.`	A field specified on the OUTPUT RECORD menu is beyond the record to be too big. size set on the GLOBALS menu. Change the GLOBALS specification or delete the field.

Field Table Errors

Error	Message	Cause/Action
301	`Internal error: OFST ALLOCATE failed.`	Space allocated for dynamic array is insufficient.
302	`Internal error: OFST record missing. Write failed previously.`	Attempt to read internal record failed because the record did not exist.
303	`Internal error: OFST write failed. (FSERR !!!!)`	Attempt to write an internal record failed.
304	`Internal error: OFST does not have any more room.`	
305	`Output field name ! not unique.`	Output field name on OUTPUT RECORD menu must be unique for the record. Use output field name to provide unique name.
306	`Internal error: OFST search range error.`	See error 301.
307	`Internal error reading OFST record. (FSERR !!!!)`	Failure to read an intermediate (internal) record.
308	`Internal error: OFST start order.`	See error 301.
309	`Internal error: OFST end order.`	See error 301.
313	`Internal error: OFST unreferenced error.`	See error 301.
316	`Internal error: OFST update screen order error.`	See error 301.
320	`Internal error: OFST wrap loop error.`	See error 301.
400	`Internal error: Field table allocation failed.`	See error 301
401	`Internal error: Field table record missing. Prev write failed.`	See error 302.
402	`Internal error reading FST from forms file. (FSERR !!!!)`	Field names in output record menu cannot be checked against the forms file because of internal inability to read information from forms file.
403	`Internal error writing FST to reformat file. (FSERR !!!!)`	See error 303.
404	`Internal error: Field table has no more room.`	See error 301.

Error	Message	Cause/Action
405	`Internal error reading input seq from reformat file. (FSERR !!!!)`	See error 402.
406	`Internal error: Input seq table record missing. Prev write failed.`	See error 302.
407	`Internal error writing seq table to refmt file. (FSERR !!!!)`	
450	`Form ! does not exist in the forms file.`	Form name specified on INPUT FORMS menu is not in forms file. Check name and correct it.
451	`Internal error reading glob forms buf from form file. (FSERR !!!!)`	Failure to read list of forms in the forms file.
452	`Internal error: Global forms file rec missing. Prev write failed.`	
453	`Internal error reading field name table for form !. (FSERR !!!)`	Failure to read list of all fields for a particular form in the forms file.
460	`Internal error in reading globals from forms file. (FSERR !!!!)`	
461	`Internal error: Allocation of form'glob'buf failed.`	
462	`Internal error: Allocation of form'flds'buf failed.`	
463	`Internal error: Allocation of literal table failed.`	
464	`Internal error: Literal table record missing. Prev write failed.`	
465	`Internal error: Error reading literal table record. (FSERR !!!!)`	
466	`Internal error writing literal table record. (FSERR !!!!)`	
468	`Internal error: Allocation of global buffer failed.`	
469	`Form ! does not exist in this forms file.`	

Menu Processing Errors

Error	Message	Cause/Action
501	`Selection field cannot be blank.`	Enter one of the following codes in the MAIN menu selection box: A, X, G, F, L, or D.
502	`Reformat identifier ! not found.`	A reformat id entered on the MAIN menu is not in the reformat file. Check the INPUT FORMS menu or correct the identifier.
503	`Output field name ! not found.`	An output field name entered on the MAIN menu is not in the reformat file. Check the OUTPUT RECORD menus or correct the field name.
504	`No reformat identifier was specified.`	A selection on the MAIN menu requires a reformat identifier. Enter the identifier or change the selection.
505	`There are no input form sequences in the deleted was specified.`	A reformat identifier was specified, but no INPUT FORMS reformat file. menu has been entered.
506	`No reformat to be deleted was specified.`	"D" was entered in MAIN menu selection box; specify the reformat identifier to be deleted.
507	`No such reformat identifier exists.`	A reformat identifier on the MAIN menu is not on any INPUT FORMS menu.
508	`Main selection field must be "A!, "X", "G", "F", "L" or "D".`	Enter one of the listed codes.
509	`This field must be blank when deleting.`	The reformat id for deletion should be in the right hand box; clear in the lefthand box.
510	`Listing is not done on a field level.`	Enter "L" to list, and a reformat identifier in the left hand box; you cannot list a single field.
511	`Form name must be blank for compile.`	When you enter "X" to compile the reformat file, you cannot enter a reformat identifier.
512	`Field name must be blank for compile option.`	When you enter "X" to compile the reformat file, you cannot enter an output field name.
513	`Form name is not specified when adding a form.`	When you enter "A" to add a reformat, do not specify the reformat id; reformat id's are entered on the INPUT FORMS menu.
514	`Fields are not added from the MAIN menu.`	Output fields are added on the OUTPUT RECORD menu, not using the "A" option of the MAIN menu.
515	`Form name must be blank for this option.`	A selection was made on the MAIN menu that cannot use a reformat id. Clear the reformat id box.

Error	Message	Cause/Action
516	`Field name must be blank for this option.`	A selection was made on the MAIN menu that cannot use an output field name. Clear the output field box.

Menu Processing Utility Errors

Errors	Message	Cause/Action
600	`NUMn can have digits only from "0" to "9".`	A value other than a digit was entered for a NUMn type field; change the data type or the selection.
606	`Must be NUMn, DIG, IMPn,MDY, DMY, YMD, or CHAR.`	Enter one of the listed codes as the data type of this field; n can be a digit 0-9 for NUM, 1-9 for IMP.
608	`Justify field must be "L", "R", "C", or " ".`	Enter one of the codes to justify field Left, Right, or Center; default is Right justify for numbers or dates, no justification for characters.
609	`Checkdigit must be either "10" or "11".`	If you want to insert a check digit, specify 10 for modulus 10, 11 for modulus 11.
610	`Checkdigit can only be added to fields of type DIGIT or CHAR.`	If you want to add a checkdigit, change the data type on the OUTPUT FIELD menu.
611	`Internal error: Substring start column value could not be displayed.`	Intrinsic ASCII failed.
612	`Numeric fields cannot be converted to fields of date type.`	When the input field is a number, you cannot specify a data type of MDY, DMY, or YMD on the OUTPUT FIELD menu. Change the data type.
613	`Date fields cannot be converted to fields of numeric type.`	When the input field is a date, you cannot specify DIG, IMPn, or NUMn as the data type on an OUTPUT FIELD menu. Change the data type to a date or CHAR.
614	`Fields of type CHAR cannot be converted to other data types.`	When the input field is type CHAR, you cannot change the data type on an OUTPUT FIELD menu. Leave the type as CHAR.
615	`Internal error: Input length value could not be displayed.`	Intrinsic ASCII failed.
617	`Only fields of type NUM, NUMn and IMP can be signed.`	If the input field is not one of the listed types, you cannot specify a sign on the OUTPUT FIELD menu.
618	`Sign must be "L", "R","F", "Z", "N" or " ".`	Only the listed codes or a space can be entered on the SIGN box of the OUTPUT FIELD menu. Correct your entry.
620	`Plus field must be "Y"or"N".`	If you want the plus sign retained in the field, enter Y in the PLUS SIGN? box; if N or blank, only minus signs are inserted.

Errors	Message	Cause/Action
621	`Expected "F", "V", or "U" in record format field.`	Enter one of the listed codes in the Output Record Format box of the GLOBALS menu; default is F (fixed length).
622	`Record size must be a numeric value less than 32767.`	Enter a positive integer less than 32767 in the Record Length box of the GLOBALS menu, or leave the default length of 80 characters.
624	`Expected literal, system constant or ASCII equivalent in string.`	Record terminator or field separator string can only be a quoted string, the numeric equivalent of an ASCII c.haracter preceded by $, or a system constant: $LF, $CR, $GS, $US, or $RS.
625	`Output field must be expanded to allow for checkdigit.`	If you enter 10 or 11 in the CDIGIT box of the OUTPUT FIELD menu, be sure to increase the size of the output field on the OUTPUT RECORD menu.
626	`Internal error: Checkdigit option not valid for data type.`	See error 610
627	`Strip fields must start and end with a quote.`	Enclose any characters in a STRIP box of the OUTPUT FIELD menu between quotes.
628	`Use "" or ’’ to indicate " or ’ inside a literal.`	Use double quotes for any quotation marks within a literal (literal is entered in the GLOBALS menu.)
629	`Plus option cannot be "Y" if sign option is "N".`	If you want a plus sign, you must also specify where you want any sign in the SIGN box of the OUTPUT FIELD menu.
630	`Fill all has already filled leading blanks.`	Enter a character in Fill ALL OR Fill Leading, not both.
631	`Fill all has already filled leading blanks.`	Enter a character in Fill ALL OR Fill Leading, not both.
632	`IMPn can have digits only from "0" to "9".`	Number of implied decimal places must be a digit.

Menu Init and Processing Errors

Error	Message	Cause/Action
702	`Internal error. Intrinsic ASCII failed.`	
703	`Largest legal integer value is 32767.`	Error in Record Length field of GLOBALS menu.
704	`Length of field cannot be negative.`	Negative value specified as field length on OUTPUT RECORD menu.
705	`Internal error initializing OUTPUT RECORD menu. (COMSTATUS !!!!)`	Internal error in writing the header part of the output record menu.
706	`Internal err writing old output recdef to screen (COMSTATUS !!!!)`	Internal error in writing user's specifications--part of output menu.
707	`Internal error reading output recdef from screen. (COMSTATUS !!!!)`	Attempt to read the output record menu failed.
708	`Internal error in writing output recdef to refmt file. (FSERR !!!!)`	
709	`Internal error reading source rec from reformat file. (FSERR !!!!)`	
710	`Internal error in writing source rec to reformat file. (FSERR !!!!)`	If trying to change the forms file name in REFSPEC make sure that the original forms file does not exist by temporarily renaming it.
714	`Internal error writing the window. (COM'STATUS !!!!)`	
716	`Internal error writing the error cursor pos'n. (COM'STATUS !!!!)`	
742	`Internal error: Start column could not be displayed.`	Intrinsic ASCII failed.
743	`Internal error: Length could not be displayed.`	Intrinsic ASCII failed.
744	`Forms file is not a valid forms file or forms file is not compiled.`	The forms file specified on the FORMS FILE menu is not valid. Correct the name or go back to FORMSPEC and compile the forms file.
745	`Internal error in T'READ'NO'HOME. Bad key read. (COMSTATUS !!!!)`	

Error	Message	Cause/Action
746	`Internal error in cursor positioning before read.`	
747	`Internal error writing the current reformat rec for !. (FSERR !!!)`	
748	`Internal error initializing Input Form Menu for reformat id !.`	
749	`Internal error reading MAIN MENU form. (COMSTATUS !!!!)`	
750	`Internal error showing MAIN MENU form. (COMSTATUS !!!!)`	
751	`Internal error getting MAIN MENU form. (COMSTATUS !!!!)`	
752	`Internal error in initializing output record menu. (COMSTATUS !!!!)`	
753	`Internal error: VSETERROR failed. (COMSTATUS !!!!)`	
754	`Internal error: FGETINFO failed.`	
755	`File is not a forms file.`	File name specified on the FORMS FILE menu does not name an existing forms file. Correct the name.
756	`File is not a reformat file.`	The file name specified on the REFORMAT file menu does not identify an existing reformat file.
757	`Internal error: Could not get information from reformat file.`	Intrinsic `FGETINFO` failed.
758	`Internal error: Could not get information from forms file.`	Intrinsic `FGETINFO` failed.

Menu Controller Errors

Error	Message	Cause/Action
802	`The PREV key is not allowed here.`	There is no previous menu.
803	`The function key just hit is not defined for this mode.`	This key has not meaning on the current menu.
805	`Error getting next form !. (COM'STATUS !!!)`	
806	`Internal error in VSHOWFORM. (COM'STATUS !!!!)`	
807	`Internal error in VREADFIELDS. (COM'STATUS !!!!)`	
808	`Internal error in VGETBUF. (COM'STATUS !!!!)`	
810	`Internal error in writing reformat rec for reformat !. (FSERR !!!)`	
811	`Internal error in reading reformat rec for reformat !. (FSERR !!!)`	
812	`Internal error in reading globals record. (FSERR !!!!)`	
815	`NEXT is not meaningful until a forms file name is specified.`	Enter a forms file name on the FORMS FILE menu before using the **NEXT** key.
816	`NEXT is not appropriate. There are no more reformats.`	Use the MAIN menu to go to a particular reformat, or use **PREV REFORMAT** key to go to a prior reformat.
817	`NEXT is not meaningful until a reformat file name is specified.`	Enter a reformat file name on the REFORMAT FILE menu before using the **NEXT** key.
818	`You cannot specify more than 255 reformats.`	You have already entered 255 INPUT FORMS menus; try to use fewer.
819	`The NEXT REFORMAT key is inappropriate.There are no more reformats.`	There are no more INPUT FORMS menus in the file.
820	`There are no previous reformats.`	The current reformat is the first one. Use the main menu to go to a particular reformat or use the **NEXT REFORMAT** key to go to the next input forms sequence.
821	`Internal error in writing globals table record. (FSERR !!!!)`	

Error	Message	Cause/Action
822	`WARNING: Changes in input forms menu may have invalidated this menu.`	The OUTPUT FIELD menu may be affected by changes to the INPUT FORMS menu.
823	`WARNING: Changes in input forms menu may have invalidated this menu.`	The OUTPUT RECORD menu may be affected by changes to the INPUT FORMS menu.
824	`Internal error: Missing record for reformat !. Prev write failed.`	
825	`NEXT'REFORMAT is not meaningful until forms file name is specified.`	
826	`NEXT'REFORMAT not meaningful until reformat file name is specified.`	
827	`Cannot go to MAIN MENU until reformat file name has been specified.`	
828	`Cannot go to MAIN MENU until forms file name has been specified.`	Do not press **MAIN** key until you have entered forms file name on FORMS FILE menu.
829	`Reformat file was modified since last compile. Press EXIT to exit.`	You pressed **EXIT** before compiling the reformat file; press **MAIN** to go to MAIN menu to compile, or press **EXIT** to exit without compiling.

Init and Compile Errors

Error	Message	Cause/Action
906	`Error writing reformat !. (FSERR !!!)`	
913	`Internal error closing your forms file. (FSERR !!!!)`	
914	`Internal error in storing the field symbol table. (FSERR !!!! )`	
917	`Internal error in storing OFST. (FSERR !!!!)`	
918	`Error closing reformat file. (FSERR !!!!)`	
919	`Internal error in closing REFSPEC forms file. (FSERR !!!!)`	
920	`Internal error in closing terminal file. (FSERR !!!!)`	
921	`Internal error in writing globals record on exit. (FSERR !!!!)`	
922	`Internal error in storing literal table. (FSERR !!!!)`	
923	`Error in opening user's forms file. (FSERR !!!!)`	Refer to the file system error to determine the cause of the open failure.

Compile Errors

Error	Message	Cause/Action
1000	`Internal error on compile. Bad field type.`	
1001	`Internal error writing field intermediate code rec. (FSERR !!!!)`	
1002	`Internal error writing reformat code record. (FSERR !!!!)`	
1003	`Internal error: Allocation of compile buffer failed.`	
1004	`Internal error writing globals code record. (FSERR !!!!)`	

Error	Message	Cause/Action
1005	`Internal error reading field intermediate code rec. (FSERR !!!!)`	
1006	`REFSPEC version has been changed. Reformat file invalid.`	The specified reformat file was produced on an earlier version of REFSPEC.
1007	`Reformat file is empty. There is nothing to compile.`	Enter A to add a reformat before entering X to compile.
1008	`Error in output record menu in reformat !.`	Go to the OUTPUT RECORD menu for the specified reformat, and make correction.
1009	`Error in input forms menu in reformat !.`	Go to the specified reformat (INPUT FORMS menu) and make correction.
1010	`Output field ! of reformat ! is in error.`	Go to the specified OUTPUT FIELD menu and make correction.
1011	`Internal error writing code rec for input forms seq. (FSERR !!!!)`	

Reformat Messages

Reformat Messages

Error	Message	Cause/Action
2	`Error opening reformat file. (FSERR !!!!)`	Check file system error number.
4	`Batch file could not be opened. (FSERR !!!!)`	Check file system error number.
5	`No reformatting was done.`	No sequence of forms in batch matched with any Reformat sequence.
6	`Batch file: !`	Not an error. Listing message.
8	`Error closing batch file. (FSERR !!!!)`	Check file system error number.
9	`Error closing output file. (FSERR !!!!)`	
10	`Error closing reformat file. (FSERR !!!!)`	
11	`Error opening output file. (FSERR !!!!)`	
13	`Cannot append to a file with variable length records.`	If you specify an existing output file in the :`FILE` command, and expect to write/append to that file, it must have fixed-length records.
14	`Reformat file not compiled or not a legal reformat file.`	Check the reformat file name in your :FILE command; you may have to run REFSPEC to compile the reformat file.
15	`Form file versions from batch and reformat file don’t match.`	A warning that forms file was recompiled since batch file was created.
16	`Batch data must be from forms file !.`	Reformat file expects batch file written from forms in specified forms file. Check that batch and reformat names are correct.
17	`Not a legal batch file.`	Check batch file name in :FILE command.
18	`REFORMAT TERMINATED ABNORMALLY.`	`FGETINFO` failed. See error message given in your output above this message.
19	`Internal error getting info from reformat file. (FSERR !!!!)`	
20	`Reformat file: !`	Not an error.
22	`Output file: !`	Not an error.

Error	Message	Cause/Action
23	`Internal error getting information from batch file. (FSERR !!!!)`	`FGETINFO` failed.
24	`File specified was not a reformat file.`	Check reformat file name in :`FILE` command.

Allocation Messages

Error	Message	Cause/Action
30	`Internal error: Batch data space too small. Allocation failed.`	
31	`Internal error allocating batch information buffer.`	
32	`Internal error allocating batch data buffer.`	
33	`Internal error allocating batch temporary buffer.`	
34	`Internal error allocating reformat code buffer.`	
35	`Internal error allocating output buffer.`	
36	`Internal error allocating temporary output buffer.`	
37	`Internal error allocating data buffer.`	

I/O Messages

Error	Message	Cause/Action
50	`Error reading batch record # !!!. (FSERR !)`	
51	`Error reading from reformat file. (FSERR !!!!)`	
52	`Error reading from reformat file. (FSERR !!!!)`	
54	`End of file reached on output file. Record not written.`	
55	`Error writing to output file. (FSERR !!!!)`	
56	`Error reading from reformat file. (FSERR !!!!)`	

Intrinsic Messages

Error	Message	Cause/Action
73	`Internal error: Translation to EBCDIC failed.`	Intrinsic failed.

Access Method Messages

Error	Message	Cause/Action
086	`Access method: file code error.`	
087	`Access method: Attempt to add a duplicate key.`	
088	`Access method: Internal error. Key insertion location error.`	
089	`Access method: Internal error. Block not present.`	
090	`Access method: Attempt to open an old file as new.`	
091	`Access method: Internal error. Key block buffer error.`	
092	`Access method: Internal error. Invalid key block number.`	
093	`Access method: Internal error. Block not verified.`	
094	`Access method: Record not found.`	
095	`Access method: Not enough space in extension for the directory.`	Fast form files require 500 words of space in the COMAREA extension. If you are coding in BASIC, list the forms file to find how many words are needed for the extension, and increase the COMAREA field USERBUFLEN to this number.
096	`Access method: Internal error. Parent block not found.`	
097	`Access method: Internal error. Illegal entry number.`	
098	`Access method: The file is not a KSAMless forms/ref file.`	
099	`Access method: The file is at EOF.`	

Code Interpretation Messages

Error	Message	Cause/Action
101	`Record terminator causes output record overflow. Data truncated.`	End-of-record marker makes record longer than output record length defined on GLOBALS menu for reformat file.

Error	Message	Cause/Action
102	`Field separator causes output record overflow. Data truncated.`	See error 101. See globals menu.
103	`Literal causes output record overflow. Data truncated.`	A literal constant in output record makes record longer than output record length defined for file in GLOBALS menu.
104	`Internal error: Bad code for reformat sequence.`	
105	`Output length greater than largest possible output record.`	Length of actual output record is longer than largest variable length record as defined on GLOBALS menu for file.
106	`Field causes output record overflow. Data truncated.`	An output field causes the output record to be longer than truncated. the record length defined on the GLOBALS menu.
107	`Field causes output record overflow. Field data not written.`	In this case, the entire field is omitted from the output record. (For error 106, the field is written with data truncated.)
108	`Field causes overwrite on previous data. Field data not written.`	The beginning of this field overlaps a previous field. data not written. Check OUTPUT RECORD specification for reformat file.
109	`System constant $Gs causes record overflow. Data truncated.`	Group separator in output record makes record longer than maximum record size defined for output file.
110	`System constant $RS causes record overflow. Data truncated.`	Record separator in output record makes record longer than Data truncated. maximum record size defined for output file.
111	`System constant $US causes record overflow. Data truncated.`	Unit separator in output record makes record longer than maximum truncated. record size defined for output file.
112	`System constant $LF causes record overflow. Data truncated.`	Line feed specified in output record makes record longer than maximum record size defined for output file.
113	`System constant $CR causes record overflow. Data truncated.`	Carriage return specified in output record makes record longer than maximum record size defined for output file.
114	`ASCII character causes output record overflow.Data truncated.`	The numeric equivalent of an ASCII character in the output record makes record longer than maximum record size defined for output file.

Statistics Messages

Error	Message	Cause/Action
200	`Reformatting completed successfully with no errors.`	Hurrah!
201	`!!!! reformats attempted.`	Message reports how many times reformat identifiers were matched with records in the reformat file.
202	` !!! records written to output file.`	This message tells you the total number of reformatted records written by REFORMAT. Message is always issued.
203	` !!! batch records processed.`	This message tells you the total number of records in the batch file that were processed by REFORMAT.
204	`!!!! errors.`	The total number of errors detected by REFORMAT.
205	`Some errors were found see above. REFORMAT terminated normally.`	This message is issued after message 204 when REFORMAT completes despite errors.

Reformat Errors

Error	Message	Cause/Action
300	`Invalid numeric data.`	Numeric data expected for output field, but data in batch field not numeric.
301	`Numeric data conversion failed.`	Conversion of numeric data from batch field to output field not successful. Check that OUTPUT FIELD is a numeric type, that batch field contains numeric data.
302	`Numeric data conversion failed. Output field too small.`	Output field defined in reformat file not large enough for numeric data in batch field.
303	`Invalid data for digit data type.`	Output field defined as type DIG, but data in batch field is not solely digits.
304	`Numeric data conversion failed.`	See error 301.
305	`Numeric data conversion failed. Output field too small.`	See error 302.
306	`Date conversion failed.`	Conversion of a date from batch field to output field not successful. Check that OUTPUT FIELD is a date type, that batch field contains a date.
307	`Date conversion failed. Output field too small.`	The output field is not large enough to contain the date in the batch field. Check field size in OUTPUT RECORD menu, check batch field.

Error	Message	Cause/Action
308	`Invalid data in date input field.`	Output field expects a date, but batch field does not contain a date.
309	`No data movement. Truncation of significant data would occur.`	The numeric data is not written to the output field because it would be truncated.
315	`Non-numeric characters cannot be converted to zone decimals.`	Output field defined as SIGN = Z, but batch field does not contain zone decimals. numeric data.
316	`No sign inserted. Truncation of significant data would occur.`	Output field defined as SIGN = F, L, P, or Z, but inserting sign would cause value in batch field to be truncated.
317	`Data truncated for output record # !!!.`	Data written to the specified output record has been truncated. Presumably, the data is not significant.
318	`No reformatting done for the following:`	The listed (fields/records) in the batch file have not written to the output file.
319	`Insertion of checkdigit would cause loss of significant data.`	The output field into which a checkdigit is to be inserted is not large enough to contain the extra digit. The checkdigit is not added.
320	`Illegal character infield. Checkdigit not generated.`	In performing the checkdigit calculations, a character other than a letter (A-Z) or a digit (0-9) was found.
321	`Internal error: More than 9 decimal places expected.`	Batch data. Involves NUM(n) field where n is valued internally to be greater than 9.
322	`All blank numeric field cannot be reformatted.`	A batch field containing blanks cannot be converted to a numeric output field.
323	`Commas in numeric data field are not correctly positioned.`	Any commas in a numeric batch field must be correct; otherwise value cannot be converted to numeric output field.
324	`Numeric data has more than the expected number of decimal places.`	Numeric batch field has more decimal digits than defined for the output field. Output field is replaced with blanks.
325	`Invalid character in numeric data field.`	Batch field to be converted to numeric output field contains characters other than digits, sign, decimal point, or commas.
326	`Invalid character in digit data field.`	Batch field to be converted to digit type output field contains characters other than digits. Output field replaced with blanks.
327	`Only a plus or minus sign was found in numeric field.`	Batch field to be converted to numeric output field contains only a sign. REFORMAT replaces the output field with blanks.

Error	Message	Cause/Action
328	`Internal error: More than 9 decimal places expected.`	See error 321.
329	`An all blank field is not a legal implied value.`	A batch field to be converted to an IMPn type output field must contain values other than blanks.
330	`Commas in implied data field are not correctly positioned.`	In an IMPn type field, commas should be positioned counting back from the implied decimal point.
331	`Implied data has more than the expected number of decimal places.`	IMPn type batch field has more than n digits to the right of an actual decimal point. Output field replaced with blanks.
332	`Illegal character found in implied data field.`	IMPn type batch field contains character other than a digit, decimal point, sign, or commas.
333	`Only a plus or minus sign was found in implied field.`	IMPn type batch field contains only a sign; cannot convert to numeric output field.
334	`IMPn field must have at least !!!! digits.`	IMPn type batch field has fewer than n digits.
335	`Illegal implied field data.`	IMPn type batch field has data other than digits, decimal point, sign, or commas.
336	`Check digit requested on empty field.`	Cannot generate check digit from blank batch field.
337	`Check digit requested on field containing only + or - sign.`	Cannot generate check digit from sign only.
338	`Check digit requested on field containing special characters.`	Can only generate check digit from digits (0-9) or letters (A-Z).
339	`Check digit 10 is invalid for modulus 11 calculation.`	This number or string is inappropriate for modulus 11 check digit calculation because that process yields a value of 10 when only single digit results can be used.
340	`Check digit modulus must be 10 or 11.`	Check OUTPUT FIELD in reformat file; only modulus 10 or 11 check digits can be generated.
341	`Check digit generation failed.`	See appendix D. Check digits cannot be calculated for some fields

Message Info

Error	Message	Cause/Action
400	`Reformat identifier: !`	
401	`Reformat field name: !`	
402	`Batch record form name: !`	
403	`Batch record #: !!!`	
404	`Output record #: !!!`	
405	`Output record col #: !!!!`	

Header Messaages

Error	Message	Cause/Action
500	`*******************************`	*********************************
501	`*`	*
502	`*`	REFORMAT HP32209! *
503	`*`	! *
504	`PAGE !!!!`	
505	`!`	
506		
507	`REFORMAT/3000 HP32209!`	

Testlist Errors

Error	Message	Cause/Action
640	`Error writing to testlist file. (FSERR !!!)`	
641	`EOF reached on testlist file.`	
642	`Error opening testlist file. (FSERR !!!)`	
643	`Error allocating testlist buffer.`	

Native Mode Access Error Messages

Error	Message	Cause/Action
996	`Forms file must be adjusted with FORMSPEC before it can be used.`	
997	`Forms file is corrupt.`	
998	`Negative argument passed to VPLUS intrinsics`	See Appendix J.
999	`Attempt to access Compatibility Mode VPLUS services failed.`	See Appendix J.

C USASCII Character Set in Collating Sequence

ASCII Character Set

Table C-1. ASCII Character Set

Hex.	Dec.	Octal Left	Octal Right	Char
00	0	000000	000000	NUL (null)
01	1	000400	000001	SOH (start of heading)
02	2	001000	000002	STX (start of text)
03	3	001400	000003	ETX (end of text)
04	4	002000	000004	EOT (end of transmission)
05	5	002400	000005	ENQ (enquiry)
06	6	003000	000006	ACK (acknowledge)
07	7	003400	000007	BEL (bell)
08	8	004000	000010	BS (backspace)
09	9	004400	000011	HT (horizontal tabulation)
0A	10	005000	000012	LF (line feed)
0B	11	005400	000013	VT (vertical tabulation)
0C	12	006000	000014	FF (form feed)
0D	13	006400	000015	CR (carriage return)
0E	14	007000	000016	SO (shift out)
0F	15	007400	000017	SI (shift in)
10	16	010000	000020	DLE (data link escape)
11	17	010400	000021	DC1 (device control 1, X-ON)
12	18	011000	000022	DC2 (device control 2)
13	19	011400	000023	DC3 (device control 3, X-OFF)
14	20	012000	000024	DC4 (device control 4)
15	21	012400	000025	NAK (negative acknowledge)
16	22	013000	000026	SYN (synchronous idle)

Table C-1. ASCII Character Set

Hex.	Dec.	Octal Left	Octal Right	Char
17	23	013400	000027	ETB (end of transmission block)
18	24	014000	000030	CAN (cancel)
19	25	014400	000031	EM (end of medium)
1A	26	015000	000032	SUB (substitute)
1B	27	015400	000033	ESC (escape)
1C	28	016000	000034	FS (file separator)
1D	29	016400	000035	GS (group separator)
1E	30	017000	000036	RS (record separator)
1F	31	017400	000037	US (unit separator)
20	32	020000	000040	blank
21	33	020400	000041	!
22	34	021000	000042	"
23	35	021400	000043	#
24	36	022000	000044	$
25	37	022400	000045	%
26	38	023000	000046	&
27	39	023400	000047	' (closing single quote)
28	40	024000	000050	(
29	41	024400	000051	)
2A	42	025000	000052	*
2B	43	025400	000053	+
2C	44	026000	000054	, (comma)
2D	45	026400	000055	-
2E	46	027000	000056	. (period)
2F	47	027400	000057	/
30	48	030000	000060	0
31	49	030400	000061	1
32	50	031000	000062	2
33	51	031400	000063	3

Table C-1. ASCII Character Set

Hex.	Dec.	Octal Left	Octal Right	Char
34	52	032000	000064	4
35	53	032400	000065	5
36	54	033000	000066	6
37	55	033400	000067	7
38	56	034000	000070	8
39	57	034400	000071	9
3A	58	035000	000072	: (colon)
3B	59	035400	000073	; (semicolon)
3C	60	036000	000074	<
3D	61	036400	000075	=
3E	62	037000	000076	>
3F	63	037400	000077	?
40	64	040000	000100	@
41	65	040400	000101	A
42	66	041000	000102	B
43	67	041400	000103	C
44	68	042000	000104	D
45	69	042400	000105	E
46	70	043000	000106	F
47	71	043400	000107	G
48	72	044000	000110	H
49	73	044400	000111	I
4A	74	045000	000112	J
4B	75	045400	000113	K
4C	76	046000	000114	L
4D	77	046400	000115	M
4E	78	047000	000116	N
4F	79	047400	000117	O
50	80	050000	000120	P

Table C-1. ASCII Character Set

Hex.	Dec.	Octal Left	Octal Right	Char
51	81	050400	000121	Q
52	82	051000	000122	R
53	83	051400	000123	S
54	84	052000	000124	T
55	85	052400	000125	U
56	86	053000	000126	V
57	87	053400	000127	W
58	88	054000	000130	X
59	89	054400	000131	Y
5A	90	055000	000132	Z
5B	91	055400	000133	[
5C	92	056000	000134	\
5D	93	056400	000135	]
5E	94	057000	000136	^ (caret)
5F	95	057400	000137	_ (underscore)
60	96	060000	000140	` (opening single quote)
61	97	060400	000141	a
62	98	061000	000142	b
63	99	061400	000143	c
64	100	062000	000144	d
65	101	062400	000145	e
66	102	063000	000146	f
67	103	063400	000147	g
68	104	064000	000150	h
69	105	064400	000151	i
6A	106	065000	000152	j
6B	107	065400	000153	k
6C	108	066000	000154	l
6D	109	066400	000155	m

Table C-1. ASCII Character Set

Hex.	Dec.	Octal Left	Octal Right	Char
6E	110	067000	000156	n
6F	111	067400	000157	o
70	112	070000	000160	p
71	113	070400	000161	q
72	114	071000	000162	r
73	115	071400	000163	s
74	116	072000	000164	t
75	117	072400	000165	u
76	118	073000	000166	v
77	119	073400	000167	w
78	120	074000	000170	x
79	121	074400	000171	y
7A	122	075000	000172	z
7B	123	075400	000173	{
7C	124	076000	000174	| (vertical line)
7D	125	076400	000175	}
7E	126	077000	000176	~ (tilde)
7F	127	077400	000177	DEL delete

D CHECK DIGIT CALCULATION

Check digits are digits added to the end of a value that contains only numbers (0-9) or letters of the alphabet (A-Z). A FORMSPEC option can be selected that causes a modulus check to be made on a value entered at the terminal.

Whenever a value is entered for which check-digit verification has been specified, a modulus calculation is performed. The result of a modulus calculation is a single digit that must match the last digit (the "check digit") of the entered value. If the match is not successful, an error is diagnosed. Note that for modulus checks to be meaningful, the last digit verified by such a check must have been calculated by the same method used for the check.

REFSPEC has an option that lets you add a check digit to an entered value. The check digit is added when REFORMAT is run. Thus, the check digit is added after the value is entered at the terminal but before it is input to the application program. In this case, the modulus calculation is made programmatically.

The modulus calculations used by VPLUS/V are either modulus 10 or modulus 11. A value is checked by one or the other of these calculations, depending on which was used to add the check digit. Thus, a value whose check digit was added using modulus 10 can be checked only by modulus 10; and a value with a modulus 11 check digit can be checked only with modulus 11 calculations.

MODULUS 10

Modulus 10 calculations detect single transpositions and incorrect keying of a single digit. The calculation is performed as follows:

- The units position and every alternate position of the basic number is multiplied by 2.
- The digits resulting from the multiplication are added to those that were not multiplied.
- The total is subtracted from the next higher number ending with zero. The result is the check digit.

Example

Assume a basic number	9 6 4 3 8
Unit and every alternate	9 4 8
Multiplied by 2	18 8 16
Digits not multiplied	6 3
Add them together	1+8+6+8+3+1+6 = 33
Next higher number ending in zero	40
Subtract sum of digits	-33
Check digit =	7
The base number with check digit	96438**7**

MODULUS 11

Modulus 11 detects single digit errors, single transpositions, and double transpositions. Unlike other check digit systems, it is based on a weighted checking factor for each digit in the basic number. The modulus 11 check digit is obtained as follows:

- Each digit position of the basic number is assigned a weighted checking factor . The following factors are assigned, starting with the units digit and progressing toward the high-order digit:

  ```
  2 3 4 5 6 7 2 3 4 5 6 7 2 3 4 . . .
  ```

- Each digit in the basic number is multiplied by its checking factor.
- The products are summed and then divided by 11. The remainder is subtracted from 11. The result is the check digit.

Example

Assume a basic number	5 1 6 1 9 2 8 7 2
Checking factors	4 3 2 7 6 5 4 3 2
Add the products	20+3+12+7+54+10+32+21+4=163
Subtract remainder from 11	11-9=2
Check digit =	2
Self-checking number	516192872**2**

If a check digit is generated using modulus 11 calculations and the result is 10, the check digit cannot be used and an error is returned. Modulus 11 check digits are the remainder from dividing the product of the calculations by 11 (see example above). Thus, if check digits are being generated for a continuous series of numbers, every eleventh number must be skipped to avoid this error.

If the product generated through the modulus 11 calculations is evenly divisible by 11 (no remainder), the resulting check digit is 11. In this case, the digit 0 is appended to the basic number.

To summarize, if the calculated check digit is 10, an error is returned; if the calculated check digit is 11, a zero is appended to the basic number.

When you attempt to add a modulus 11 check digit that evaluates to 10, the reformatter issues the message: "Check digit is invalid for modulus 11 calculation".

If ENTRY checks a field according to the FORMSPEC statement `CDIGIT 11` and that field contains a value with a check digit that evaluates to `10`, the same message is issued.

ALPHABETIC CHECK DIGITS

Letters of the alphabet are treated like numbers in either modulus 10 or modulus 11 check digit calculations. This is done by assigning a digit to each letter of the alphabet as follows:

A	B	C	D	E	F	G	H	I	J	K	L	M	N	0	P	Q	R	S	T	U	V	W	X	Y	Z	(=space)
1	2	3	4	5	6	7	8	9	1	2	3	4	5	6	7	8	9	2	3	4	5	6	7	8	9	0

A value to be checked may be preceded by a plus or minus sign; however, the sign is ignored in the check digit calculations. To illustrate, perform a modulus 10 check digit calculation on a value that contains letters as well as numbers:

Assume a basic value	`3 4 G 1 2 H`
Change the letters to digits	`3 4 7 1 2 8`
Multiply alternate digits by 2	`8 2 16`
Digits not multiplied	`3 7 2`
Sum with digits that were not multiplied	`3+8+7+2+2+1+6 = 29`
Subtract sum from next higher number ending with zero	`30-29 = 1`
Check digit =	`1`
Self-checking number	34G12H**1**

E Application Hints

This appendix provides hints and guidelines on the following topics

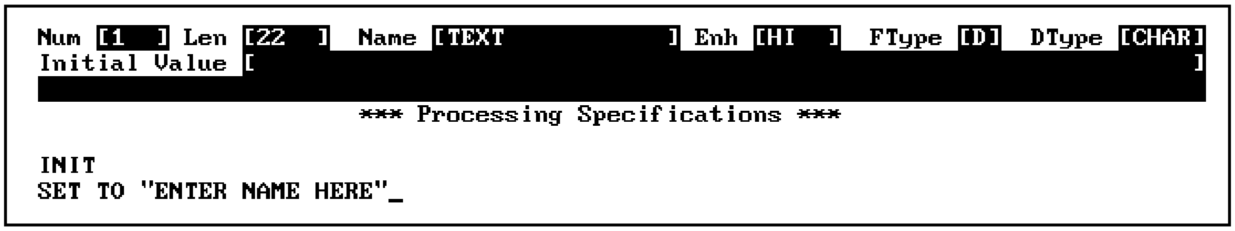
- Designing your Forms File
- Rules for the DL Area
- Coding the Touch Feature

Designing Your Forms File

The following hints may help you design your forms file with FORMSPEC:

1. If the status/window line is the last line on the screen, it should be no longer than 79 characters. This is because using the 80th character on the bottom line of the screen causes the screen to roll up one line temporarily. If this roll-up is not a problem, then the last line can use 80 characters.
2. On an HP 264X terminal, when a form is to be appended, column 80 of any line of the form with an unprotected field should not be used. When the next form is displayed, column 80 of the current form disappears.
3. If you want to enhance text (the protected area of the screen), you can define a field with a display only field type and any enhancements you desire. Using the INIT processing phase specification, text can be initialized to appear in the field. For example:

   This method can also be used to alter "text" on the screen during execution. Define a display only field with the desired enhancements. You can either use the processing specifications phases (`INIT, FIELD` or `FINISH`) to alter the contents of the field or you can programmatically change the field contents using VPUTBUFFER or VPUTFIELD intrinsic calls.



```
Num [1   ] Len [22   ]  Name [TEXT              ] Enh [HI   ]  FType [D]  DType [CHAR]
 Initial Value [                                                                   ]

                             *** Processing Specifications ***

 INIT
SET TO "ENTER NAME HERE"_
```

NOTE Processing specifications only execute if the application calls the "editing" intrinsics: `VINITFORM, VFIELDEDITS, VFINISHFORM`.

4. If you want to define blank lines at the bottom of a form, define a display only field of at least one character at the end of the blank lines. Delimit the field by **ESCAPE** and **ESCAPE** , with the field enhancements of NONE. This forces the display of seemingly blank lines and does not affect the application programs. Otherwise FORMSPEC would delete trailing blank lines from a form.
5. If you use save fields to accumulate totals and you expect the user to browse and correct the fields used for summation, special care must be taken to ensure that the totals are accurate. If an entered value is summed into a save field, and then this value is changed in browse/modify mode, the new value is also summed into the save field unless you specify the SET statement to account for this possibility.

   For example, assume you are accumulating values entered in a field `F1` into a save field used for batch totals BT. In order to allow only the correct values to be accumulated, construct a display-only field `OLDF1` identical in its characteristics to `F1`. The field is initialized to zero in collect mode only, but not in browse mode. In browse mode, `OLDF1` contains the previous value of `F1`. This previous value is then subtracted from the sum

BT. The following specifications are entered in the F1 Field Menu to ensure that only the correct values are summed:

```
INIT

    SET OLDF1 TO 0                        \executed only in collect mode

 FIELD

    <any edit statements>

    SET BT TO BT + F1 - OLDF1          \executed in browse or collect
                                        mode
    SET OLDF1
```

6. If you want to include a logical AND function in your processing specifications, use nested IF statements. For example, to get the effect of

   ```
   IF A=B AND C=D AND E=F
   ```

   use the following statements:

   ```
   IF A=B THEN

      IF C=D THEN

         IF E=F THEN
   ```

   To negate an IF condition, simply use the ELSE part with a null THEN part.

   A logical OR or NOR cannot be similarly specified, but for comparisons on a single field, use IN or NIN with a list. For example, to get the effect of

   ```
   IF A=B OR A=D OR A=F
   ```

   use the edit statement:

   ```
   A IN B, D, F
   ```

7. Sometimes in data entry applications, the user would like to knowingly enter a value that normally falls the designer's edits. This is called "Edit Override", and can be implemented through FORMSPEC in a number of ways.

   a. Adopt a user convention to require some special character(s) in the field along with the desired data. (Note that the field must be made long enough to accommodate the extra character(s).) For example:

      ```
      IF MATCH ?*!! THEN                 \Skip the edits.

      ELSE<edit statements>               \No trailing “!”; apply normal edits
      ```

      Note that the special characters must satisfy the default data type edit. The above statement, for instance, would not work with a numeric field. Some possibilities of special characters for numeric fields are leading zeros (MATCH O?*), a leading plus (MATCH !+?*), or a comma (MATCH 000,?*). If signed numbers are expected, prefix your pattern with !+,-.

   b. Include an auxiliary EDIT OVERRIDE field that must be marked in order to bypass edits. For example, assume a field named EO:

```
    IF EO MINLEN 1 THEN

    ELSE<edit statements>                EO not marked; apply edits
```

8. The contributed utility program RESTORE will not correctly copy records larger than 2000 bytes. If you wish to copy a KSAM forms file or reformat file from a store tape to a disc file, use the MPE :RESTORE command and the FCOPY subsystem.

9. There are three file equations for which VPLUS checks during execution. The use of these file equations provides the user with a way to override the defaults used by VPLUS. These file equations are also useful when attempting to debug an application with trace messages or when DEBUG is used. The VPLUS intrinsic VOPENTERM, used by FORMSPEC, ENTRY, and REFSPEC, searches for a file equation named A264X. VOPENTERM attempts to open the device referenced by A264X to display VPLUS forms and accept their associated input. For example:

   ```
   :FILE A264X; DEV=28
   :RUN ENTRY.PUB.SYS
   ```

   causes logical device 28 to be opened by VOPENTERM, and subsequent screen displays by VSHOWFORM appear on LDEV 28. In this example, LDEV 28 must be a logged off terminal in an available state. Applications can make use of this feature by using the *termfile* parameter of VOPENTERM and a file equation. If the second parameter to VOPENTERM contained the value TERMINAL, the following file equation will cause logical device 28 to be opened for forms display:

   ```
   :FILE TERMINAL;DEV=28
   ```

   Trace and abort messages will be displayed to the command terminal — the one which initiated the application. This would be the main reason to use this feature as abort messages are frequently impossible to read because they are displayed in the unprotected fields of a VPLUS application.

   The second file equation which VPLUS searches for is FORMLIST. The VPRINTFORM intrinsic uses FORMLIST to list forms (as in ENTRY). FORMSPEC also uses FORMLIST for the LIST command. FORMLIST defaults to the system printer. If, however, the user wishes to save the list file on disc, the following file equation does just that, creating a file named FORMLIST containing the form listing.

   ```
   :FILE FORMLIST; DEV=DISC;SAVE
   ```

   If your system has both a page printer and a line printer (default), the file equation:

   ```
   :FILE FORMLIST;DEV=PP
   ```

   sends the form listing to the page printer.

   The third file equation which VPLUS uses deals with a summary listing of the forms file from batch mode FORMSPEC. Batch mode prints to a file named FORMOUT. A file equation for FORMOUT can use the same parameters as discussed for the file FORMLIST.

10. The LIST command and the batch mode FORMS command print a maximum of 60 lines per page. This maximum may be altered by setting the JCW.

    ```
        FORMSPECLINESPERPAGE
    ```

For example, the MPE command

```
:SETJCW FORMSPECLINESPERPAGE=45
```

sets the maximum number of lines per page to 45. This JCW is useful when more white space is desired at the bottom of the page, or when special paper is used.

FORMSPECLINESPERPAGE must be in the range 1 to 150. Any value outside this range will cause a LIST command to fail, and will cause batch mode FORMSPEC to halt with the error message.

```
FORMSPECLINESPERPAGE must be between 1 and 150
```

No lines will be printed in either case, to ensure that FORMSPEC does not waste paper printing a listing of FORMS reference in an unwanted format.

Rules for the DL Area

VPLUS makes use of the DL area for buffers and internal tables. Therefore, if the DL area is put to any other use during the execution of a VPLUS application, such use must not conflict with that of VPLUS. To insure against conflict, observe the following rules:

1. The first use of the DL area by an application, including any procedures, routines, or intrinsics executed on behalf of the application, should be the call to `VOPENFORMF` that opens the forms file and allocates the DL area needed by VPLUS.
2. The call to `VCLOSEFORMF` that frees up the DL area used by VPLUS must not occur until the application has released all other DL space allocated subsequent to the `VOPENFORMF` call.

These restrictions need not be followed if the application is wholly coded in any combination of Pascal, HPFORTRAN 77, and HPBUSINESS BASIC *and* sets the *comarea* item *language* to 5, which causes VPLUS to use the heap procedures provided by Pascal instead of managing the DL area directly.

VPLUS applications coded in other languages need to take great care if they pass the *language* identifier of 5 as a means of getting around the restrictions stated above. Specifying an incorrect *language* identifier can cause other aspects of the VPLUS interface, such as addressing of byte parameters, to function unreliably. In such a case, the application must use the INTRINSIC mechanism (CALL INTRINSIC in COBOL) for the main and all interacting parts of the application. Check the Pascal and COBOL reference manuals for more information.

In all cases, check for any other use of the DL area by intrinsics such as the DSG intrinsics.

Coding the Touch Feature

In order to use the touch feature effectively, a method is needed to provide feedback to the user when a field is touched. Changing the field enhancement is one of the best methods to indicate that a field has been touched. The enhancement of a field can be changed by using the VCHANGEFIELD intrinsic, or family forms. If neither of these methods is appropriate and the error count is not important, VSETERROR can be used instead. VSETERROR toggles the error flag when called successively; i.e., if VSETERROR is called the second time for a particular field, the error flag is cleared. Therefore, the error enhancement can be used with VSHOWFORM as feedback to the user to indicate that a field has been touched. This feature is only activated if the *showcontrol* bit 0 is set to 1.

When designing touch applications, the designer should keep the idea of compatibility in mind. Applications should be able to run on HP 262X/239X terminals, or on a touch terminal/workstation using keyboard input. For VPLUS applications, this objective can be achieved by designing one single form interface for all HP 262X/239X terminals.

To design a touch application with compatibility in mind, the interface should be built so that the user has the option of responding via the keyboard or the touch screen. This consideration is important in two respects. First, it provides an alternative path for the user when an application is run on touch terminals. Second, an application can be run on non-touch terminals as well as touch terminals.

For example, if an application is designed to display a form with many items for the user to select from, the programmer may wish to include one input field for the users to type in their choice. This will allow for one interface for both touch and non-touch terminals.

Example

```
IF COM-LASTKEY < 0 THEN
           CALL "VSHOWFORM" USING COMAREA
           CALL "VREADFIELDS" USING COMAREA
           IF COM-LASTKEY IS NOT EQUAL TO -999 THEN
                   NEXT-FIELD = -(COM-LASTKEY)
                   CALL "VSETERROR" USING COMAREA, NEXT-FIELD, MSG, MSGLEN
                   CALL "VSETERROR" USING COMAREA, PREVIOUS-FIELD, MSG, MSGL
EN
                   PREVIOUS-FIELD = NEXTFIELD.
```

The first VSETERROR toggles the error flag of the new field touched, and the second VSETERROR toggles the error flag of the previous field touched. When VSHOWFORM is called, NEXT-FIELD will be highlighted with the error enhancement, while PREVIOUS-FIELD reverts to the normal enhancement. The VPLACECURSOR intrinsic can be used to position the cursor if the default cursor position set by VSETERROR needs to be overridden. @

F STATE/POSTAL CODES

The special system constant $STATE consists of a table of all the state codes for the 50 states of the United States, plus U. S territories, armed services and other valid postal codes. When an entered postal code is matched against the table, the entered code can be in any combination of upper or lowercase letter. For instance the code for California is `CA`; any of the following codes can be successfully matched against the California code: `CA`, `ca`, `Ca`, or even `cA`.

The state codes listed below are shown only in uppercase for convenience.

Area Americas	AA	Mississippi	MS
Area Europe	AE	Missouri	MO
Area Pacific	AP	Montana	MT
Alabama	AL	Nebraska	NE
Alaska	AK	Nevada	NV
American Samoa	AS	New Hampshire	NH
Arizona	AZ	New Jersey	NJ
Arkansas	AR	New Mexico	NM
California	CA	New York	NY
Canal Zone	CZ	North Carolina	NC
Colorado	CO	North Dakota	ND
Connecticut	CT	Northern Mariana Islands	CM
Delaware	DE	Ohio	OH
District of Columbia	DC	Oklahoma	OK
Florida	FL	Oregon	OR
Federated States of Micronesia	FM	Palau	PW
Georgia	GA	Pennsylvania	PA
Guam	GU	Puerto Rico	PR
Hawaii	HI	Rhode Island	RI
Idaho	ID	South Carolina	SC
Illinois	IL	South Dakota	SD
Indiana	IN	Tennessee	TN
Iowa	IA	Texas	TX

Kansas	KS	Utah	UT
Kentucky	KY	Vermont	VT
Louisiana	LA	Virginia	VA
Maine	ME	Virgin Islands	VI
Marshall Islands	MH	Washington	WA
Maryland	MD	West Virginia	WV
Massachusetts	MA	Wisconsin	WI
Michigan	MI	Wyoming	WY
Minnesota	MN		

G TERMINAL INFORMATION

This appendix contains information about terminals that affects the operation of VPLUS. An overview of the supported terminals and their main features is followed by information on terminals, grouped according to terminal family and model within the family. This appendix also contains information about:

- The VPLUS *comarea* items that relate to terminals
- Configuring terminal buffers
- Recovering from unexpected program interruption
- The user environment control file
- Advanced terminal I/O procedures; VTURNON/VTURNOFF

SUPPORTED TERMINALS AND FEATURES

VPLUS can be used with the following terminals:

• HP 150*	• HP 2392A	• HP 2622A	• HP 2640B	• HP 3075A
• HP 2382A	• HP 2393A	• HP 2623A	• HP 2641A	• HP 3076A
	• HP 2394A	• HP 2624A	• HP 2642A	• HP 3081A
	• HP 2397A	• HP 2624B	• HP 2644A	
		• HP 2625A	• HP 2645A	
		• HP 2626A	• HP 2647F**	
		• HP 2627A	• HP 2648A	
		• HP 2628A		
* HP 150 is obsolete and not supported for this release. ** HP2647F opt series 6x/7x consoles are not compatible.				

Applicable special features of these terminals are shown in the table above and are described below.

Termtypes

As a user of an HP 3000 system, you may choose to specify a *termtype* when you log onto MPE; for example,

```
:HELLO username.acctname;term=nn
```

However, VPLUS only supports a subset of the available *termtypes*. They are *termtypes* 10, 12, 14, and 24.

A *termtype* that is not supported by VPLUS may be:

- Rejected as an error if VPLUS knows that it cannot continue, or
- Mapped to a supported *termtype* if this will allow VPLUS to continue.

When VPLUS maps an unsupported *termtype* to one that is supported, the terminal driver is reconfigured to the original *termtype* when VPLUS closes the terminal. Finally, VPLUS defers to the Workstation Configurator when editing *termtype* if that facility is available.

Modified Data Tag

You do not have to take any action to utilize this feature. Only the fields which have been modified are transmitted to the computer. This feature is supported on the following terminals:

HP 150 (obsolete and not supported)	HP 2393A HP 2394A HP 2397A	HP 2624A/B HP 2625A HP 2628A

Extended Local Edits

This feature provides for the editing of information by the terminal as it is typed into the form. You specify the editing to be done by using the command LOCALEDITS in the field processing specification section. The HP 2624B and HP 2394A terminals supports this feature.

Relabeling Function Keys

This feature allows you to specify during forms creation or modification time function key labels that will be used during data entry. Function key labeling is supported by VPLUS on all terminals except the data capture devices. On HP 264X terminals, VPLUS uses the last two screen lines to emulate function key labels, which are available as a terminal feature on all other HP terminals.

Security Display Enhancement

This feature allows you to inhibit the display of the data in certain fields. When this feature is used, the characters entered at the terminal are displayed as blanks on the screen. The security display enhancement is supported on the following terminals:

HP 150 (obsolete and not supported)	HP 2393A HP2394A	HP 2624B HP 2625A HP 2626A HP 2628A

Table G-1. Terminals Supported by VPLUS

ID	*	MODEL	GRAPH	COLOR	TOUCH	LFS	MDT	LOCAL EDITS	SEC ENH	X.25 PAD
0	*	%								
1		2640B								
2		2644A								
3		2645A								
4		2641A								
5		2648A	x							
6		2647A	x							
7	*	2621A								
8		2626A				x			x	
9		2624A					x	x	x	
10		2642A								
11		2622A								
12		2623A	x							&

Table G-1. Terminals Supported by VPLUS

ID	*	MODEL	GRAPH	COLOR	TOUCH	LFS	MDT	LOCAL EDITS	SEC ENH	X.25 PAD
13		2624B				x	x	x	x	&
14		2382A								&
15		3075A								
15		3081A								
16		3076A								
17	+	3077A								
18	*	2621B								
31		2647F	x							
51		2703A	x							
52		2627A	x	x						x
55		2622E								
56		2392A							x	x
57		2394A				x	x	x	x	x
61		2625A					x		x	x
62		2628A					x		x	x
63		2625A	x				x		x	x
64		2628A	x				x		x	x
65		2393A	x				x		x	x
66		2397A	x	x			x			x
67		2393A	x		x		x		x	x
68		2397A	x	x	x		x			x
70	*	HP150	x		x		x		x	x
		700/92							x	

```
* Not a supported terminal
% Unknown
+ Can only be opened and closed by VPLUS
& Check ROMS for auto keyboard lock
```

Local Form Storage Capabilities

This feature allows forms to be stored locally in the terminal's memory. Local form storage can reduce the data communication overhead associated with frequently displayed forms.

This feature is supported on the HP 2394A, HP 2624B and HP 2626A terminals, and is fully described in Section 6.

X.25 Capability

This feature allows X.25 block mode to be used via a PAD interface with the following terminals:

HP 150 (obsolete and not supported)	HP 2392A	HP 2622A*	HP 2625A
	HP 2393A	HP 2623A*	HP 2627A
	HP 2394A	HP 2624B*	HP 2628A
	HP 2397A		
* Check ROMS for auto keyboard lock.			

Color Enhancement

This feature allows color to be used for field, error, and window enhancements with the HP 2397A and HP 2627A terminals.

Data Capture Device Special Features

The HP 3075A, HP 3076A and HP 3081A terminals are data capture devices. Special features of the data capture devices that VPLUS supports are as follows:

Terminal Input Options	Terminal Output Options
-Type V Badge Reader	-Alphanumeric Printer
-Multifunction Reader	-Prompting Lights
-Magstripe Reader	-Alphanumeric Display
-Bar Code Reader	-Mini-CRT
-Standard Keyboard	
-Alphanumeric Keyboard	

The HP-IB and RS232 Input/Output options, the Numeric Display terminal, Card Image Data and Multifield Input options are not supported.VPLUS.

Cursor Position Sensing

The cursor position sensing function includes three new intrinsics that have been designed to enable application programs to sense the cursor position on a VPlus screen:

`VARMSCP`	Arms or disarms cursor sensing capability.
`VGETSCPFIELD`	Returns information about the cursor position by field number on a VPlus screen.

`VGETSCPDATA`	Returns information about the cursor location by field number and row and column number on a VPlus screen.

These intrinsics enable application programs to determine where the cursor was left on a VPlus screen after a read operation. The information returned to the application varies with the intrinsic used.

Usage

These intrinsics should be called in an application at certain points of the transaction flow:

- Call `VARMSCP` at the *`beginning`* of **each** VPlus transaction where cursor information is required. Typically, this is prior to a call to `VSHOWFORM` and preceding `VREADFIELDS`.
- Call `VGETSCPFIELD` or `VGETSCPDATA` at the *`end`* of a VPlus transaction to collect the cursor position or location information. This is *after* a call to `VREADFIELDS`.

NOTE Important Details! Please Read.

- Cursor position sensing is available on DTC-connected terminals only.
- These intrinsics work only on terminals that have the capability to return cursor location information. These terminals are:
  `700/9X` 2392A
  2394A

THE HP 264X TERMINALS

The HP 264X terminal family is the terminal default — you do not need to access FORMSPEC's Terminal/Language Selection Menu *unless* you need to change the default. However, if you want function key labeling for HP 264X terminals, you must access the Terminal/Language Selection Menu and specify Y in the HP 264X Family field. Also, when designing forms to run on these terminals, the following constraints must be considered:

- Do not use column 79 if the form is to be part of a form family.
- Do not use column 80 if the form may have another form appended to it or is part of a form family.
- A multiline field must go all the way through column 80 in order to continue on the next line.
- The security display enhancement is not supported on HP 26 4X terminals.

All HP 264X terminals, except the HP 2640A and HP 2644A terminals described below, run VPLUS with the default terminal strapping. However, if the defaults have been altered, you must *close* straps G and H. These terminals are automatically configured by VPLUS so you do not have to press the block mode key.

HP 2640B Terminal

This terminal must be placed in block mode. When the terminal is opened by FORMSPEC, ENTRY, REFSPEC, or an application, a message is displayed asking you to press the block mode key. Also, when the terminal is closed, a message reminds you to unlatch the block mode key. The HP 2640B must be strapped as follows to use VPLUS:

Switch	Setting
A	closed
B	------
C	closed
D	open
E	open
F	open
G	open
H	closed

Strapping the terminal in this manner allows BLOCK MODE/PAGE operation and allows you to use the function keys without also using **CONTROL**. The backtab feature, which allows you to use **CONTROL TAB** to position the cursor to the beginning of the previous field, is not available on this terminal.

HP 2644A Terminal

This terminal must be placed in block mode. When the terminal is opened by FORMSPEC, ENTRY, REFSPEC, or an application, a message is displayed asking you to press the block mode key. Also, when the terminal is closed, a message reminds you to unlatch the block mode key. The HP 26 44A must be strapped as follows to use VPLUS:

Switch	**Setting**
A	closed
B	------
C	closed
D	open
E	------
F	------
G	open
H	closed

Strapping the terminal in this manner allows `BLOCK MODE/PAGE` operation.

THE HP 262X TERMINALS

VPLUS runs on the HP 262X family without any special action by you (without specifying the HP262X, HP239X Families on FORMSPEC's Terminal/Language Selection Menu) *except* in the special cases noted in the following paragraphs.

To use X.25 block mode via a PAD interface, available with all HP 262X terminals except the HP 2624A, the transmit and receive pacing must be set correctly, the appropriate ROMS must be used, and the terminal must be strapped correctly. You must check that the correct ROMS are used with the HP 2622A, HP 2623A, HP 2624B and the HP 2627B terminals. The HP 262SA and HP 2628A terminals always have the correct ROMS. Also, the correct terminal type must be used when logging on to the terminal. For more detailed information, refer to the *DSN/X.25 HP 3000 Reference Manual*.

HP 2624A Terminal

To utilize the local edits and security display enhancement features of the HP 2624A, you must specify the HP262X, HP239X Families on FORMSPEC's Terminal/Language Selection Menu.

At run time, when using a form, you could receive the message:

```
LINE IS FULL - RETURN TO CLEAR.
```

This message is issued by the terminal, not by VPLUS. It means that there are too many characters on the line, and you must redesign and recompile the form.

HP 2624B Terminal

To utilize the local edits and security display enhancement features of the HP 2624B, you must specify the HP262X, HP239X Families-on FORMSPEC's Terminal/Language Selection Menu.

This terminal also has the capability of storing as many as 255 forms locally in terminal memory depending on the size of the forms and the memory available. Use of this feature is activated and controlled by the VPLUS intrinsics and *comarea* items as discussed in Section 6.

To use VPLUS and local form storage on this terminal, you must alter the default datacomm configuration menu. Change the strip NULL and DEL field from YES to NO.

At run time, when using a form, you could receive the message:

```
LINE IS FULL - RETURN TO CLEAR.
```

This message is issued by the terminal, not by VPLUS. It means that there are too many characters on the line, and you must redesign and recompile the form.

HP 2625A, HP 2628A Terminals

To utilize the security display enhancement of these terminals, you must specify the HP262X, HP239X Families on FORMSPEC's Terminal/Language Selection Menu.

HP 2626A Terminal

To utilize the security display enhancement feature of the HP 2626A, you must specify the HP262X, HP239X Families on FORMSPEC's Terminal/Language Selection Menu.

The HP 2626A terminal can store as many as four forms locally depending on the size of the forms. VPLUS utilizes the workspace/window feature of this terminal and configures terminal memory into from one to four workspaces. Local form storage is activated and controlled by the VPLUS intrinsics and *comarea* items as discussed in Section 6. Only port I of the HP 2626A terminal can be utilized with local form storage. If you wish to run an application on port 2 of the terminal without local form storage, the *formstoresize* item of the *comarea* should be modified as described in Section 6 in the *comarea* item description.

HP 2627A Terminal

To utilize the color display enhancement feature of the HP 2627A, you must specify both HP262X, HP239X Families and HP2627A, HP2397A on FORMSPEC's Terminal/Language Selection Menu.

THE HP 239X AND HP 150 TERMINALS

VPLUS runs on the HP 2 39X and the HP 150 terminals without any special action by you (without specifying the HP262X, HP239X Families on FORMSPEC's Terminal/Language Selection Menu) *except* in the special cases noted in the following paragraphs.

To use X.25 block mode via a PAD interface, available with all HP 239X and HP 150 terminals, the transmit and receive pacing must be set correctly and the terminal must be strapped correctly. (The HP 239X and HP 150 terminals always have the correct ROMS.) Also, the correct terminal type must be used when logging on to the terminal. For more detailed information, refer to the *DSN/X.25 HP 3000 Reference Manual*.

HP 2392A Terminal

To utilize the security display enhancement feature of the HP 2392A, you must specify the HP262X, HP239X Families on FORMSPEC's Terminal/Language Selection Menu.

HP 2393A Terminal

To utilize the security display enhancement feature of the HP 2393A, you must specify the HP262X, HP239X Families on FORMSPEC's Terminal/Language Selection Menu.

HP 2394A Terminal

To utilize the local edits and security display enhancement features of the HP 2394A, you must specify the HP262X, HP239X Families on FORMSPEC's Terminal/Language Selection Menu

This terminal also has the capability of storing as many as 255 forms locally in terminal memory depending on the size of the forms and the memory available. Use of this feature is activated and controlled by the VPLUS intrinsics and *comarea* items as discussed in Section 6.

HP 2397A Terminal

To utilize the color display enhancement feature of the HP 2397A, you must specify both HP262X, HP239X Families and HP2627, HP2397A on FORMSPEC's Terminal/Language Selection Menu.

HP 150 Terminal (Obsolete)

To utilize the security display enhancement feature of the HP 150, you must specify the HP262X, HP239X Families on FORMSPEC's Terminal/Language Selection Menu.

THE DATA CAPTURE DEVICES

To use the HP 3075A, HP 3076A or HP 3081A devices with VPLUS, you must specify the HP307X Family on FORMSPEC's Terminal/Language Selection Menu. To take advantage of the HP 3081A features, you must specify an eight (8) in the selection box for the HP307X Family on the Terminal/Language Selection Menu. VPLUS only opens or closes the HP 3077A device; no other functions are available.

TERMINAL COMMUNICATION AREA

Positions 49 through 58 (relative to 1) of the VPLUS communication area (*comarea*) are used for terminal-related information, which includes the terminal file number, terminal type, terminal allocation and error logging, and terminal options. It is not necessary for you to access this area as VPLUS manages it automatically. However, if you need to know the MPE file number of your terminal, how to suppress messages, read data automatically, or specify a timeout value for reading keys, the *comarea* items described in the following paragraphs perform these functions.

Parameters

Position 49 - *filen* MPE file number used to identify the terminal.

Position 56 - *termoptions* Terminal control options:

`bits: 0 - 8`	Reserved for system use.
`9 - 10`	01 = Enable **ENTER** or function key timeout in VREADFIELDS.
	00 = Disable **ENTER** or function key timeout in VREADFIELDS (default).
`13 - 14`	01 = Enable AUTOREAD in VREADFIELDS.
	00 = Disable AUTOREAD in VREADFIELDS (default)
`15`	0 = Display mode message (default)
	1 = Suppress mode message.

Normally, there is no time limit for VREADFIELDS, whether the intrinsic is terminated by **ENTER** or a function key. Timeouts can be enabled in applications by setting bit 10 of the *termoptions* item of the *comarea* to 1, and specifying a timeout interval in the *usertime* item of the *comarea*. When a timeout is enabled, if **ENTER** or a function key is not pressed within the number of seconds specified in *comarea* position 58, the *cstatus* of 160 will be returned to the *comarea*. At this point, control will then be passed to the statement following the VREADFIELDS call in the application.

For example, in COBOL:

```
INITIALIZE-COMAREA
    MOVE ZERO TO COM-OPTIONS.
    ADD 32 TO COM-OPTIONS.
    MOVE 60 TO COM-USERTIME.
```

if included in the initialization section of the application will indicate that all calls to VREADFIELDS should timeout after 60 seconds. Note that an ADD instruction was used to turn on the timeout feature. All changes to the *termoptions* item following VOPENTERM should be careful to set only single bits, as VPLUS uses bits 0-7. A MOVE instruction will destroy the configuration of these bits.

When VREADFIELDS is terminated by a function key, VPLUS does not transmit any data entered on the screen to the user buffer. If the programmer wishes for data to be transmitted to the data buffer when a function key is pressed, the AUTOREAD feature should be enabled. The following COBOL example shows how to enable the autoread feature, and transfer data to the buffer when the function key **f4** is pressed:

```
CALL "VREADFIELDS" USING COMAREA.
IF COM-LASTKEY = 4 THEN
      ADD 2 TO COM-OPTIONS
      CALL "VREADFIELDS" USING COMAREA
      SUBTRACT 2 FROM COM-OPTIONS.
```

Note that two VREADFIELDS calls are necessary to complete the process. The first VREADFIELDS detects that a function key was pressed, the second call actually transmits the data to the data buffer. The second call to VREADFIELDS does not wait for the user to press enter, but rather sends an "ESC d" to the terminal to simulate an automatic enter. As noted above in the description of user timeouts, the COBOL SUBTRACT and ADD statements should always be used when setting bits in the *termoptions* item of the *comarea*.

The mode message is displayed on terminals **not** automatically configured by VPLUS. The mode message for VOPENTERM is BLOCK MODE/PAGE IS SET; for VCLOSETERM it is REMEMBER TO UNLATCH THE BLOCK MODE KEY.

Position 58 - *usertime* If enabled, the value in this word is used as the number of seconds to wait for the **ENTER** or function key to be pressed. The maximum timeout value is 3 2,7 6 7 seconds (9.1 hours).

Position 59 - *identifier* VPLUS identifier for the HP terminal model being used, as shown earlier in Table G-1. in the column labeled ID.

TERMINAL BUFFER CONFIGURATION

When using FORMSPEC, ENTRY, REFSPEC, or the intrinsics `VREADFIELDS` or `VSHOWFORM`, there must be sufficient terminal buffers available for all concurrently executing terminal I/O operations. The number of terminal buffers must be at least 150 and it is strongly recommended that the number of terminal buffers be set to the maximum allowed on your system, shared by all processes. (See the configuration dialogue in the *MPE V Systems Operation and Resource Management Reference Manual*.

RECOVERING FROM UNEXPECTED PROGRAM INTERRUPTION

In case of unexpected interruption due to hitting **BREAK** or a terminal power failure, control returns to MPE. To recover from such a situation, you take the following steps:

1. Perform a hard reset.
    - On an HP 264X terminal, you press the **RESET TERMINAL** key twice, and then press **RETURN** to display the colon prompt,
    - On HP 262X, HP 239X, and HP 150 terminals, you press **SHIFT CONTROL RESET** and then press **RETURN** to display the colon prompt.
2. If echo is turned off, press **ESCAPE**, then press **:** to restore echo.
3. Type RESUME and you should see the message `READ PENDING`.
4. Press **REFRESH** to return to the menu at which you were interrupted.

The menu will be cleared to initial or previously entered values. To insure against damage to the file, reenter the information on all menus pertaining to the form you were creating, modifying, or entering data at the time of interruption.

If you were in the unformatted portion of a FORMSPEC Field Menu, the procedure is slightly different, as follows:

1. Perform a hard reset, as described above.
2. Type **CONTROL** `R ABORT` **CONTROL** `R` (Letters must be upper case as shown).
3. If echo is turned off, press **ESCAPE**, then press **:** to restore echo.
4. Issue the `RUN` command for FORMSPEC to continue operation.

USER ENVIRONMENT CONTROL FILE

The user environment control file is called VENVCNTL.PUB.SYS, and is designed to let the programmer assert limited control over terminal activities. It is opened and read on the VOPENTERM call, and it controls these two items in the user environment.

1. Abbreviating a terminal query. When VOPENTERM is called, it usually goes through a complicated process to identify the terminal type. The VENVCNTL file shortens the identification process. However, it should only be used if it is known that a query will elicit a valid terminal ID. To shorten the terminal query, enter 1 in the first column of the VENVCNTL file.
2. Enabling **AIDS**/**MODES**/**USERKEYS** without a program. Instead of setting the SHOWCONTROL word of the user's *comarea* in the application, enter 1 in the second column of the VENVCNTL file. This option is very useful if you do not have the source code for the software but you want to enable **AIDS**/**MODES**/**USERKEYS**.

To implement a user environment control file, build this file:

```
:BUILD VENVCNTL.PUB.SYS;DEV=DISC;REC=-80,1,F,ASCII
```

Use any editor to enter 1 in column 1 and/or 2. When VOPENTERM executes, it looks for the VENVCNTL file. If it does not find it, VPLUS carries out its usual operations. If the file exists and columns 1 and/or 2 are set, VPLUS performs either or both of the prescribed actions.

The VENVCNTL file need not necessarily by in PUB.SYS. However, if you have the file residing in another group and account, you must set up the file equation:

```
:FILE VENVCNTL.PUB.SYS=VENVCNTL.mygroup.myaccount
```

ADVANCED TERMINAL I/O PROCEDURES

In general, the VPLUS terminal input/output procedures such as `VOPENTERM`, `VCLOSETERM, VREADFIELDS` and `VSHOWFORM` are sufficient for most applications that communicate with a user via a terminal interface.

In building complex user interfaces, however, application designers have a limited number of tools for implementation. Hence, they resort to using `FREAD, FWRITE`, and input/output verbs of their programming language to display and retrieve information from the terminal when running VPLUS. The situation becomes more complicated when the application controls the terminal settings as well as the driver settings, and is likely to result in an application that runs in a restricted environment; for example, on an ATP an on one kind of terminal only.

VPLUS provides two additional procedures that help application designers to write applications with complex user interfaces that are more portable and easier to maintain. They are `VTURNON` and `VTURNOFF`.

VTURNON/VTURNOFF

VOPENTERM configures the driver and the terminal for block mode access. VCLOSETERM configures the terminal and driver for character mode access. These procedures initialize the terminal in many ways, including clearing the screen image. VTURNON and VTURNOFF reconfigure the terminal and driver but leave the terminal screen image intact. VTURNOFF allows the application to switch momentarily to character mode from block mode without disturbing the screen. A call to VTURNON will reconfigure the terminal back to block mode. An example of such use is an application which performs graphics input/output after the terminal has been opened by VOPENTERM and a form has been displayed using VSHOWFORMF. VTURNOFF can then be called to prepare for graphics operations. A subsequent VTURNON call will switch the application back to VPLUS block mode. The use of VOPENTERM, VCLOSETERM, VTURNON and VTURNOFF provides for a systematic approach to controlling the terminal and driver settings.

The recommended practice for programming VPLUS applications is to use only VPLUS procedures to perform terminal input/output operations when in VPLUS. If the application must perform terminal input/output operations using other input/output methods, either VCLOSETERM or VTURNOFF must be used to switch out of VPLUS before doing so.

This simple programming guideline assures that the application will be compatible with the different HP3000 drivers and terminals, and improves its maintainability.

VTURNOFF

Turns off VPLUS block mode and enables character mode access without disturbing the terminal screen.

Syntax

```
    VTURNOFF {comarea}
```

Parameters

comarea Must be *comarea* named when the file was opened by VOPENTERM. If not already set, the following *comarea* items must be set before calling VTURNOFF.

cstatus	Set to zero.
comarealen	Set to total number of 2-byte words of *comarea*.

VTURNOFF may set the following *comarea* items:

cstatus	Set to nonzero value if call is unsuccessful.
fileerrnum	Set to file error code if there is an MPE file error.

Discussion

VTURNOFF is used for momentarily switching from VPLUS block mode to character mode. This procedure is designed for use after a terminal has been previously opened by VOPENTERM or after a VTURNON.

VTURNOFF reconfigures the terminal and driver for character mode operations without disturbing the screen image on the terminal. The following operations normally performed in VCLOSETERM are not performed in VTURNOFF:

- Clear local form storage
- Enable the USER/SYSTEM keys
- Disable touch reporting, delete touch fields
- Clear screen
- Unlock keyboard
- Close terminal file

VTURNOFF

NOTE VTURNOFF does not close the terminal file. To close the field and completely reset the driver and the terminal, VCLOSETERM must be used.

Example

```
The following examples illustrate a call to VTURNOFF:

COBOL
CALL "VTURNOFF" USING COMAREA.

BASIC
200 CALL VTURNOFF (C(*))

FORTRAN
CALL VTURNOFF (COMAREA)

SPL
VTURNOFF (COMAREA);

PASCAL
VTURNOFF (COMAREA);
```

VTURNON

Turns on VPLUS block mode without disturbing the terminal screen.

Syntax

```
VTURNON {comarea,TERMFILE}
```

Parameters

comarea The *comarea* name must be unique for each open forms file. The *comarea* must be the same one used in VOPENTERM. The following *comarea* items must be set before the call if not already set:

cstatus	Set to zero.
language	Set too code that identifies the programming language of the calling program.
comarealen	Set to total number of 2-byte words in *comarea*.

VTURNON may set the following *comarea* fields.

cstatus	Set to nonzero value if call is unsuccessful.
fileerrnum	Set to file error code if there is an MPE file error.
filen	Set to MPE file number of terminal.
identifier	Set to appropriate VPLUS terminal ID.
lab'info	Set to appropriate number and length of labels.

Discussion

VTURNON is normally used in an application when the terminal has already been opened by VOPENTERM and VTURNOFF was called to switch out of VPLUS block mode. VTURNON switches the application back to VPLUS block mode without disturbing the image on the terminal screen.

After a VTURNON, you must perform a VSHOWFORM. You must also unlock your keyboard. Sending the escape sequence **ESC-b** unlocks the keyboard programmatically, or depressing the **RESET** key once also unlocks the keyboard.

VTURNON reconfigures the terminal and the driver without performing the following operations, which are usually carried out by VOPENTERM.

- Initialize local form storage
- Clear screen
- Enable the **USERKEYS** function keys
- Disable or enable the USER/SYSTEM key as specified in the SHOWCONTROL word

Unlike VOPENTERM, VTURNON will not ask you to press the BLOCK MODE key if you are using an HP2640B or HP2644 terminal and the terminal is not on block mode.

Example

The following examples illustrate a call to VTURNON:

H Version Control

This appendix provides information on:

- KSAM File Control and Conversion to MPE Files
- Adjust Menu
- The HP32209 intrinsic
- The HP32209B utility

KSAM File Management

Since the release of version B.03.23 of VPLUS, VPLUS accepts only MPE files with the file code of VFORM. VPLUS has created only MPE files since VPLUS version A.01.01. Prior to A.01.01, VPLUS created KSAM files. Any KSAM file created by VPLUS before version A.01.01 will continue to function when accessed through FORMSPEC, REFSPEC, ENTRY or your applications without any actions on your part. If, however, you wish to modify the forms file using FORMSPEC or REFSPEC, the file must be converted using the CONVERT utility, as described below.

Purge Forms File

The MPE :PURGE command will delete a non-KSAM forms file. If, however, your forms file is a KSAM file, MPE does not provide a direct means to purge it. If you want to permanently delete a KSAM forms file created through a prior version of FORMSPEC, you must run the KSAMUTIL utility program as follows:

```
:RUN KSAMUTIL.PUB.SYS
```

This program will issue a greater than sign (>) as a prompt. In response to this prompt, you enter the PURGE command. Suppose the forms file you want to purge is named ORDFORM, enter the command as follows:

```
>PURGE ORDFORM
ORDFORM.PUB.ACCTG & ORDKEY PURGED  system response
```

Note that you need not specify the key file name associated with your forms file; KSAMUTIL knows this name as it indicates in the response to the PURGE command. KSAMUTIL also notes the account and group in which the forms file resides.

Rename Forms File

If you want to rename an existing non-KSAM forms file, simply use the MPE :RENAME command. In order to rename a KSAM forms file, you must run program KSAMUTIL. You may rename either the forms file or the key file with a single RENAME command; or you may rename both files with two RENAME commands. Suppose you want to rename the forms file ORDFORM and also its associated key file ORDKEY. To do this, run KSAMUTIL and then enter two RENAME commands, as shown:

```
:RUN KSAMUTIL.PUB.SYS

>RENAME ORDFORM, NEWORD
                         (new names)
>RENAME ORDKEY, NEWKEY
```

Convert KSAM File

Before converting a KSAM file to the now standard MPE files, you should first use FCOPY to physically remove any records which have been logically deleted in the file. This can be accomplished as follows:

```
:FCOPY FROM=ORDFORM;TO=(ORDFORM2,ORDKEY2)
```

This will copy only the active records in the KSAM file named ORDFORM to the new KSAM file named ORDFORM2 whose key file in ORDKEY2, also a new file.

The utility program `CONVERT.PUB.SYS` will create an MPE file from the KSAM file created above:

```
:RUN CONVERT.PUB.SYS
PLEASE ENTER NAME OF THE OLD FORMS FILE.
ORDFORM
PLEASE ENTER THE NEW FORMS FILE NAME.
ORDNEW
DIRECTORY LENGTH=1264 - system response
END OF PROGRAM
```

The directory is found in an extension to the *comarea*, and consists of information regarding the location of the data records in the forms file. Its length varies from forms file to forms file and is indicated in the system response as in the example.

Adjust Menu

The Adjust Menu is automatically displayed from the Forms File Menu the first time a forms file created before VPLUS version B.03.03 is accessed from FORMSPEC version B.03.03 or beyond.

You are given a choice of exiting `FORMSPEC` by pressing f8, or continuing with the adjustment process by entering `CONTINUE` into the only field on the Adjust Menu. It is recommend that you create a backup copy of the forms file before the adjustment. Once the forms file is converted to the B.03.03 (or later) format, it is not compatible with VPLUS version prior to B.03.03.

HP32209

Returns the version identifier for the installed version of the VPLUS intrinsics.

Syntax

```
HP32209 {buffer}
```

Parameters

buffer	Fourteen-byte character string defined in an application to which the version identifier is returned.

Discussion

The version identifier is in the format `HP32209`*`v.uu.ff`* where *`v`* denotes the major enhancement level, *`uu`* the update level, and *`ff`* the fix level.

Example

```
COBOL
CALL INTRINSIC HP32209 USING BUFFER

BASIC
200 CALL HP32209(B$)

FORTRAN
CALL HP32209 (BUFFER)

SPL/PASCAL
HP32209(BUFFER)
```

The HP32209B Utility

The HP32209B utility provides you with a simple way to confirm the version of the VPLUS software that you are currently running on the system. HP32209B resides in the PUB.SYS group and account. You can use it by typing the MPE command

```
:RUN HP32209B.PUB.SYS
```

The screen will display the product name and version as well as the version number of each segment of the software. This is to indicate any patches that may have a version number different from the product.

Refer to Appendix J for more information on the use of the MPE/XL version of this utility, called HP32209S, with HP Precision Architecture.

Example

```
:HP32209B

VPLUS INTRINSICS VERSION:  HP32209B.06.08

      ------- --------------
      SEGMENT    VERSION
      ------- --------------
        01    B’06’08’S01’00
        02    B’06’08’S02’00
        03    B’06’08’S03’00
        04    B’06’08’S04’00
        05    B’06’08’S05’00
        06    B’06’08’S06’00
        07    B’06’08’S07’00
        08    B’06’08’S08’00
        09    B’06’08’S09’00
        10    B’06’08’S10’00
        11    B’06’08’S11’00
        12    B’06’08’S12’00
        13    B’06’08’S13’00
        14    B’06’08’S14’00
        16    A.00.06.S16.00
        17    B’06’08’S17’00
        18    B’06’08’S18’00
        19    B’06’08’S19’00
        20    B’06’08’S20’00
        21    B’06’08’S21’00
        22    B’06’08’S22’00
      ------- --------------
        DHCF      A4004000

VPLUS/Windows VERSION: HP36393A.00.06
```

I BIT MAP CONVERSION

A procedure called `BITMAPCNV` is provided to help decode and set bits in programming languages which do not have bit operations. It is particularly helpful in `COBOL` with the bit map returned by `VGETFIELDINFO`.

Syntax

```
BITMAPCNV {bitmap,bytemap,numbytes,function,cnverr}
```

Parameters

bitmap	Source bit array containing bits to be converted to a byte array, or destination bit array for conversion from a byte array.
bytemap	Destination or source byte array. (For COBOL use BYTEMAP)
mbytes	Two-byte integer set to number of bytes to be converted from or to.
function	Two-byte integer set to conversion type; • `1` = bit to byte, • `2` = byte to bit
cnverr	Two-byte integer variable containing conversion error return; • `0` = conversion successful, • `-1` = conversion failed, • `-2` = invalid input parameter

Discussion

This procedure can be used to set bits in the *comarea*. It can also be used to decode the error bit map returned by VGETFORMINFO. The BITMAPCNV procedure decodes a bit map to a byte array if the *function* parameter contains a value of one. Each byte in the byte array represents the corresponding bit in the bit map. If a bit in the bit map is on (value of one), the corresponding byte will contain the character of one (1). If the bit is off, the byte will contain the character of zero (0).

If the *function* parameter passed contains a value of two, the conversion will be from byte to bit. A bit map will be produced to reflect the 1's and 0's in a byte array.

When converting bytes to bits, anything in a byte other than the character zero (0) causes the corresponding bit in the bit map to be set to one (1).

In COBOL, BITMAPCNV can be called as shown in the example below.

Example

```
CALL "BITMAPCNV" USING BITMAP,@@BYTEMAP,NUMBYTES,FUNCTION,CNVERR.
```

J HP PRECISION ARCHITECTURE

This appendix offers some guidelines for the migration of programs developed on MPE/V to MPE/XL systems, and a short list of peripherals and languages supported on MPE/XL systems.

It also provides information on those aspects of VPLUS that pertain exclusively to MPE/XL running on the HP Precision Architecture (HPPA) systems. They are:

- The intrinsics `VGETIEEEREAL/VGETIEEELONG` and `VPUTIEEEREAL/VPUTIEEELONG`
- The HP32209S utility

MIGRATION ISSUES

Programs that were originally developed on MPE/V must be rewritten to run in native mode on the MPE/XL. For details, refer to the following reference manuals:

- *Migration Process Guide* describes how to migrate existing applications to MPE/XL.
- *Programmers' Skills Migration Guide* explains new program development on MPE/XL and serves as a reference to the operating system.
- *COBOL II/XL Migration Guide* explains how to migrate COBOL II/V programs to MPE/XL.
- *FORTRAN 77/XL Migration Guide,* details the migration of FORTRAN 77/V applications to MPE/XL.
- *HP Pascal/XL Migration Guide* details the migration of Pascal applications to MPE/XL. Migration of SPL programs to MPE/XL.

Some points to be noted when applications developed on VPLUS are run MPE/XL are described below.

Terminals. When you run an application on MPE/XL, the terminal must be configured for the XON/XOFF handshake. If it is not, large or complex forms will appear to be corrupted when displayed on the screen.

Floating Point Numbers. The type conversion of floating point numbers in VPLUS assumes the HP 3000 floating point format, whereas native mode applications store floating point values in IEEE format. The type conversion will work, but the values will be wrong.

Native Mode Pascal Applications. The MPE/XL native mode Pascal compiler allocates 4 bytes for the integer subrange -32768..32767, compared with the 2 bytes allocated by the MPE/V pascal compiler. You should use type SHORTINT in native mode Pascal applications to get a 2-byte allocation for this subrange.

Record padding in native mode Pascal will cause data misalignment problems for programmers using the VPLUS intrinsics, VGETFIELDINFO, VGETFORMINFO, and VGETFILEIINFO. The problem can be avoided by packing the *inforbuf* records passed as parameters.

Incorrect Parameters. Existing VPLUS applications written in COBOL and recompiled into MPE/XL native mode may fail to function. The application either fails with VPLUS error code of 998 or 999, or it aborts with a Memory Fault error. The reason for this is that the programmer has coded the application to pass a display numeric data type as the parameter for a VPLUS intrinsic, instead of a binary numeric data type. That is, COBOL data type PIC 9 has been coded instead of PIC 9 COMP.

This kind of mistake is not caught when the application is compiled, because most COBOL VPLUS applications do not use the compiler intrinsic mechanism, "CALL INTRINSIC...USING...". So, the error must be caught at run-time, if at all. It is possible for the defective program to appear to work correctly in MPE/V and MPE/XL compatibility mode, because VPLUS is more forgiving when parameters of the wrong type are passed to it. However, native mode applications on MPE/XL call the VPLUS intrinsics through switch stubs, which require the correct parameter types. The wrong parameter causes a run-time error because the VPLUS native mode access layer does not have the capacity to edit the data passed through it.

If your application fails with a VPLUS error 998 or 999, or a memory fault, you can take the following steps:

1. Activate the VPLUS trace facility and examine the trace, especially the entering and leaving messages from the VPLUS native mode stub software.
2. Check the parameter content messages of the last VPLUS intrinsic call before the program failed.
3. Examine and correct VPLUS parameter declarations as needed, both in native mode code and in compatibility mode code if this is a migrated application.

Supported Peripherals and Applications

```
The following are currently supported on MPE/XL.

Printers-HPIB 2565, 2566, 2680, 2688

Terminals 2392, 2393, 2394, 2397,
2622, 2624B, 2627,
    HP 150, Vectra, Port+

Native Mode COBOL II/XL, FORTRAN 77/XL
Languages HP Pascal/XL

Compatibility Mode SPL/V RPG/V, Business BASIC/V,
Languages               Transact/V
```

HPPA INTRINSICS

Introduction

On the MPE/V and previous systems, Hewlett Packard represented real or floating point numbers in a format unique to each machine. However, on the MPE/XL systems, these numbers are represented in a format that conforms with an IEEE standard. This means that programs running in native mode on MPE/XL systems will deal with numbers stored in the IEEE format. VPLUS will be running in compatibility mode, however, and will thus deal with numbers stored in the MPE/V format.

Currently, VPLUS offers intrinsics that will accept a real number n the MPE/V format, convert it to its equivalent external representation and put it into a form's data buffer. The four HPPA intrinsics described below have been developed to allow programs running in native mode to perform the same operations on a number stored in the IEEE format.

NOTE Use these intrinsics only if you are programming entirely in native mode. If you are accessing a database in compatibility mode, you must use the VPLUS intrinsics, `VGET/PUTtype`.

VGETIEEEREAL

Copies character coded data contents from data buffer into an application, converting numeric value to IEEE floating point format.

Syntax `VGETIEEEREAL {`*`comarea,fieldnum,variable`*`}`

Parameters *`comarea`* Must be the *`comarea`* name specified when the forms file was opened with `VOPENFORMF`. If not already set, the following *`comarea`* items must be set before calling `VGETIEEEREAL`:

`cstatus`	Set to zero.
`comarealen`	Set to total number of two-byte words in *`comarea`*.

`VGETIEEEREAL` may set the following *`comarea`* items:

`cstatus`	Set to nonzero value if call is unsuccessful or if requested field has an error.

`fieldnum`	Two byte integer variable containing the number assigned to the field by FORMSPEC. The field itself may not be longer than 80 characters.
`variable`	Variable within application, of type IEEE floating point real (32 bits), into which converted value is placed.

For Example:

```
VGETIEEEREAL(COMAREA,FIELDNUM,VARIABLE);
```

Discussion The content of the field denoted by the field number is read from the data buffer. This content must be numeric but its data type need not be; numbers in a field of

CHAR type can be converted.

NOTE The field number is a unique number assigned to each field by FORMSPEC and is completely independent of the field's position in the data buffer or the form. It only changes if the batch command, RENUMBER, is used. You can alter the length, position or other characteristics of a field without affecting the field number. The field number provides a way to locate fields regardless of their position.

The numeric value, stored in the buffer in character coded form, is converted to IEEE floating point format and then copied to the variable in the application. If errors occur during conversion, *cstatus* is set to the appropriate error code. If the requested field's error code is set, its value is moved to the variable but *cstatus* is set to an error code.

VGETIEEELONG

Copies character coded data contents from data buffer into an application, converting numeric value to IEEE floating point long format (64-bit).

Syntax `VGETIEEELONG {`*`comarea,fieldnum,variable`*`}`

Parameters *comarea* Must be the *comarea* name specified when the forms file was opened with VOPENFORMF. If not already set, the following *comarea* items must be set before calling VGETIEEELONG:

cstatus	Set to zero.
comarealen	Set to total number of two-byte words in *comarea*

VGETIEEELONG may set the following *comarea* items:

cstatus	Set to nonzero value if call is unsuccessful or if requested field has an error.

fieldnum Two-byte integer variable containing the number assigned to the field by FORMSPEC. The field itself may not be longer than 80 characters.

variable Variable within application, of type IEEE floating point (64 bits), into which converted value is placed.

For example:

```
VGETIEEELONG(COMAREA,FIELDNUM,VARIABLE);
```

Discussion The content of the field denoted by the field number is read from the data buffer. The content of the field must be numeric, but its data type need not be; numbers in a field of CHAR type can be converted.

NOTE The field number is a unique number assigned to each field by FORMSPEC and is completely independent of the field's position in the data buffer or the form. It only changes if the batch command, RENUMBER, is used. You can alter the length, position or other characteristics of a field without affecting the field number. The field number provides a way to locate fields regardless of their position.

The numeric value, stored in the buffer in character coded form, is converted to IEEE floating point long format and then copied to the variable in the application. If errors occur during conversion, *cstatus* is set to the appropriate error code. If the requested field's error code is set, its value is moved to the variable but *cstatus* is set to an error code.

VPUTIEEEREAL

Writes a floating point number in IEEE standard format from an application to a specified field in the form data buffer, converting the value to character set coded external representation.

Syntax VPUTIEEEREAL {*comarea,fieldnum,variable*}

Parameters *comarea* Must be the *comarea* name specified when the forms file was opened with VOPENFORMF. If not already set, the following *comarea* items must be set before calling VPUTIEEEREAL:

cstatus	Set to zero.
comarealen	Set to total number of two-byte words in *comarea*.

VPUTIEEEREAL may set the following *comarea* items:

cstatus	Set to nonzero value if call is unsuccessful.
numerrs	May be decremented if a new value replaces the old value of a field with an error.

fieldnum Two-byte integer variable containing the number assigned by FORMSPEC to the field in which the value is written. The field itself may not be longer than 80 characters.

variable Variable within application, of type IEEE floating point real (32 bits), that contains the value to be converted to character set coded external representation and copied to a field in the data buffer.

For example:

```
VPUTIEEEREAL(COMAREA,FIELDNUM,VARIABLE);
```

Discussion This intrinsic converts an IEEE floating point real number to its character coded form and writes the converted value to a particular field in the data buffer, right justified. The exact format of the written data depends on the type of destination field. For example, if the number "34.56" were put to a field of type DIG, the result would be "34" since such a field may contain only the digits 0 through 9. The destination field is identified by the field number assigned by FORMSPEC, and must be defined as a numeric field, that is, type NUM, IMP or DIG.

If the specified field had an error, VPUTIEEEREAL will clear the field's error flag and decrement *numerrs*.

NOTE The field number is a unique number assigned to each field by FORMSPEC and is completely independent of the field's position in the data buffer or the form. It only changes if the batch command, RENUMBER, is used. You can alter the length, position or other characteristics of a field without affecting the field number. The field number provides a way to locate fields regardless of their position.

VPUTIEEELONG

Writes a floating point number in IEEE standard long format from an application to a specified field in the form data buffer, converting the value to character set coded external representation.

Syntax VPUTIEEELONG {*comarea*,*fieldnum*,*variable*}

Parameters *comarea* Must be the *comarea* name specified when the forms file was opened with VOPENFORMMF. If not already set, the following *comarea* items must be set before calling VPUTIEEELONG:

cstatus	Set to zero.
comarealen	Set to total number of two-byte words in *comarea*.

VPUTIEEELONG may set the following *comarea* items:

cstatus	Set to nonzero value if call is unsuccessful.
numerrs	May be decremented if a new value replaces the old value of a field with an error.

fieldnum Two-byte integer variable containing the number assigned by FORMSPEC to the field to which the value is written. The field itself may not be longer than 80 characters.

variable Variable within application, of type IEEE floating point long (64 bits), that contains the value to be converted to character set coded external representation and copied to a field in the data buffer.

For example:

```
   VPUTIEEELONG(COMAREA,FIELDNUM,VARIABLE);
```

Discussion This intrinsic converts an IEEE floating point long number to its character coded form and writes the converted value to a particular field in the data buffer, right justified. The exact format of the written data depends on the type of the destination field. For example, if the number "34.56" were put to a field of type DIG, the result would be "34" since such a field may contain only the digits 0 through 9. The destination field is identified by the field number assigned by FORMSPEC, and must be defined as a numeric field, that is, NUM,IMP or DIG.

If the specified field had an error, VPUTIEEELONG will clear the field's error flag and decrement *nuerrs*.

NOTE The field number is a unique number assigned to each field by FORMSPEC and is completely independent of the field's position in the data buffer or the form. It only changes if the batch command, , is used. You can alter the length, position or other characteristics of a field without affecting the field number. The field number provides a way to locate fields regardless of their position.

THE HP32209S UTILITY

The HP322095 utility provides you with a simple way to confirm the versions of both the native mode software and the compatibility mode software that you are currently running on the system. HP322095 resides in the PUB.SYS group and account. You can use it by typing the MPE command.

```
:RUN HP32209S.PUB.SYS
```

The screen will display the product name and version as well as the version number of each segment of the software. This is to indicate any patches that may have a version number different from the product.

This is a quick way to test whether native mode is moving through to compatibility mode on the MPE/XL system, and hence to confirm that the VPLUS software is accessible in compatibility mode.

Example When you run HP32209S, the screen displays the information in the format shown in the following example, first for native mode and then for compatibility mode. In this example, VPLUS native mode is enabled and the VPLUS switch to compatibility mode is disabled.

```
:HP32209S

VPLUS INTRINSICS VERSION:  HP32209B.06.08

      ------- --------------
      SEGMENT    VERSION
      ------- --------------
        01    B’06’08’S01’00
        02    B’06’08’S02’00
        03    B’06’08’S03’00
        04    B’06’08’S04’00
        05    B’06’08’S05’00
        06    B’06’08’S06’00
        07    B’06’08’S07’00
        08    B’06’08’S08’00
        09    B’06’08’S09’00
        10    B’06’08’S10’00
        11    B’06’08’S11’00
        12    B’06’08’S12’00
        13    B’06’08’S13’00
        14    B’06’08’S14’00
        16    A.00.06.S16.00
        17    B’06’08’S17’00
        18    B’06’08’S18’00
        19    B'06'08'S19'00
        20    B'06'08'S20'00
        21    B'06'08'S21'00
        22    B'06'08'S22'00
      ------- --------------

VPLUS NATIVE MODE INTRINSICS VERSION:  HP32209B.06.08
                     OVERLAY VERSION:  B.06.08.00


FOR THIS USER:

   VPLUS NATIVE MODE IS ENABLED;
   VPLUS SWITCH TO COMPATIBILITY MODE IS DISABLED;
   WILL DEFAULT COMPATIBILITY MODE SL SEARCH TO
   SYSTEM SL

        DHCF      A4004000

VPLUS/Windows VERSION:  HP36393A.00.06

VPLUS/Windows Native Mode VERSION:  HP36393A.00.06
                  OVERLAY VERSION:  A.00.06.00
```

K SNA DHCF with VPLUS Applications

This appendix provides advisory information about running VPLUS applications from IBM 3270 display stations using the Systems Network Architecture Distributed Host Command Facility (SNA DHCF). Note that the IBM 3270 Display System features several display station models. Consequently, this appendix generically refers to this family of display stations as IBM 3270.

The sections in this appendix provide information about:

- The purpose and function of SNA DHCF and its impact on VPLUS
- IBM 3270 screen and keyboard characteristics that affect VPLUS
- The effect of SNA DHCF on VPLUS intrinsics
- The effect of SNA DHCF on FORMSPEC
- The effect of SNA DHCF on other VPLUS applications

For additional information on these topics, refer to the *SNA DHCF Application Programmer's Manual*.

SNA DHCF Overview

SNA DHCF enables IBM System 370-compatible mainframes and series 900 HP 3000 systems to communicate interactively. Users of IBM 3270 display stations with access to the IBM Host Command Facility (HCF) can access an HP 3000 in an SNA network and initiate a session on the MPE XL operating system. SNA DHCF converts ASCII data from HP 3000 VPLUS and TTY-compatible applications to the EBCDIC format that IBM 3270 display stations require, and vice versa.

To avoid potential confusion, it is important to note that IBM has a counterpart to Hewlett-Packard's SNA DHCF, which is called DHCF.

System Connections

An IBM user can execute HCF commands to access an application on a remote IBM minicomputer. If an IBM user wants to access an application on a remote HP 3000 instead, he or she would use HCF and HP SNA DHCF, rather than IBM DHCF. HP SNA DHCF emulates IBM DHCF, enabling the connection of an HP 3000 to an IBM mainframe. When SNA DHCF is used, an HP 3000 perceives an IBM 3270 display station to be a virtual terminal (VT) device. Conversely, an IBM mainframe perceives an HP 3000 to be a remote IBM minicomputer.

VPLUS Supportability

SNA DHCF supports VPLUS applications if you recompile forms files for use with IBM 3270 display stations. Certain IBM 3270 display station limitations affect full-screen VPLUS applications. For instance, the expanded HP terminal feature set offers display enhancements, such as inverse video, that are unavailable on many IBM 3270 display stations. Consequently, you cannot display these enhancements on an IBM 3270 display station when using SNA DHCF.

Applications must exclusively use VPLUS intrinsics for all terminal I/O between `VOPENTERM` and `VCLOSETERM` calls, because the SNA DHCF subsystem is in unedited mode when performing VPLUS I/O. Unedited mode means that the subsystem expects valid 3270 data streams (in EBCDIC) from the VPLUS intrinsics.

If you design HP 3000 application programs with VPLUS-based forms for IBM 3270 application users, or for both HP 3000 and IBM 3270 users, it is imperative to note the IBM 3270 display station and forms compilation limitations described in subsequent sections of this appendix.

IBM 3270 Differences and Limitations

When using VPLUS to design or modify applications for IBM 3270 users, it is important to understand the limitations that restrict forms design for IBM 3270 display stations. The information in the following sections explain how an IBM 3270 differs from an HP 3000 terminal and the impact of these differences when running VPLUS applications using SNA DHCF.

Keyboard Differences

Table K-1 compares HP 3000 terminal keys with equivalent IBM 3270 keys.

Table K-1. Comparison of Major Keys

HP 3000 Key	Equivalent IBM 3270 Key and Explanation
AIDS	No equivalent key exists.
BREAK	**PA1 -** Equivalent to a system break. A VPLUS application can disable this key. If disabled, SNA DHCF ignores the key when pressed and re-enables the keyboard.
CLEAR DISPLAY	**CLEAR -** Functions as a refresh key by clearing the entire screen, including data and the form itself. Consequently, this key is not a local function, like the equivalent HP key.
CLEAR LINE	**ERASE EOF -** Clears data in an unprotected field from the cursor to the end of the field, regardless of the number of lines in the field. This differs slightly from **CLEAR LINE**, which performs the same function only if the field does not exceed one line.
CTRL	No equivalent key exists.
CTRLY	**PA2 -** Equivalent to a subsystem break. A VPLUS application can disable this key. If disabled, SNA DHCF ignores the key when pressed and re-enables the keyboard.
DEL LINE	No equivalent key exists.
ENTER	**ENTER -** Locks the keyboard and transmits all modified data. The cursor remains in the same position.
ESC	No equivalent key exists.
INS LINE	No equivalent key exists.
MODES	No equivalent key exists.
NEXT PAGE	No equivalent key exists.
PREV PAGE	No equivalent key exists.
RESET	**RESET -** Unlocks a locked keyboard.
ROLL DOWN	No equivalent key exists.

Table K-1. Comparison of Major Keys

HP 3000 Key	Equivalent IBM 3270 Key and Explanation
ROLL UP	No equivalent key exists.
USER KEYS	No equivalent key exists.

In addition to the keys listed above, IBM 3270 function keys differ from HP 3000 function keys. The IBM keyboard has 12 or 24 function keys; the first 8 function keys map to HP keyboards. Table K-2 shows the mapping of HP function keys to IBM 3270 function keys.

Table K-2. Mapping of Keys

HP Function Key	IBM 3270 Function Key
ENTER (0)	ENTER
f1 (1)	PF1, PF13
f2 (2)	PF2, PF14
f3 (3)	PF3, PF15
f4 (4)	PF4, PF16
f5 (5)	PF5, PF17
f6 (6)	PF6, PF18
f7 (7)	PF7, PF19
f8 (8)	PF8, PF20
ENTER (0)	PF9, PF21
ENTER (0)	PF10, PF22
ENTER (0)	PF11, PF23
ENTER (0)	PF12, PF24
BREAK	PA1
CTRLy	PA2
REFRESH	CLEAR

Screen Differences

SNA DCHF affects the screen design of VPLUS applications for IBM 3270 display stations in the following areas:

- Screen attributes
- Screen enhancements
- Function key labels
- Fields

- Character translation

Screen Attributes

IBM 3270 display stations have the following screen attributes and limitations:

Display Area	Depending on the IBM 3270 display station model, the display area is either 24, 32, or 43 lines long. The maximum length for a VPLUS form is 24 lines. Consequently, if you display a VPLUS form on an IBM 3270 that has 32 or 43 lines, the lines following line 24 will be blank. SNA DHCF reserves the last two lines of the display area for function key labels. Therefore, a form can consist of either 21 lines including a window line, or 22 lines without the window line.
Status Line	A status line appears below the display area. SNA DHCF applications, including VPLUS, cannot access the status line. Only the display controller can update this line.
Window Line	A window line defined in a forms file consumes a line of the available display. If you define a window line in the forms file, you must place the window line on line 1 or line 22, because IBM 3270 limitations prohibit inserting a line in the middle of a form. Selecting a window line reduces the form length to 21 lines. You must reserve a fixed line if you define a window line in a forms file, because window lines cannot be inserted. SNA DHCF limits the text in a window line to 79 bytes instead of 80, because space must be reserved for the attribute byte. Text exceeding 79 bytes becomes truncated. (Refer to the section entitled "Fields" for information about the attribute byte.)
Modified Data Tag	An IBM 3270 functions as a modified data tag (MDT) device. It transmits to the HP 3000 only the data that the end user has modified.
Limitations	The following IBM 3270 limitations are important to note when designing or modifying VPLUS applications. The IBM 3270 does not have: • Forms caching capability • Terminal memory other than display memory, which consists of a 24-line length and 80-column width • Scrolling capability In addition, the IBM 3270 does not support: • The line-drawing character set provided on some HP

terminals

- Local field edits that occur in the terminal firmware, such as edits for numeric fields

Screen Enhancements

Several HP terminal display enhancements are unavailable on the IBM 3270. You do not have to alter any enhancements specified in an existing forms file for IBM 3270 usage, however.

Table K-3 summarizes the mapping of HP display enhancements to IBM 3270 display stations. The codes provided for some of the HP display enhancements correspond to the value specified in FORMSPEC for the display enhancements of fields, errors, and the window line. Note that the half-bright enhancement (code=H) overrides all other enhancements except for security.

Table K-3. Display Enhancements

HP Display Enhancement	IBM 3270 Display Enhancement
Security (code=S)	Non-display
Half-bright (code=H)	Normal display
No enhancement (code=NONE)	Intensified display
Blinking (code=B)	Intensified display
Underlining (code=U)	Intensified display
Inverse video (code=I)	Intensified display
Color pair 1 (red/black)	Intensified display
Color pair 2 (green/black)	Intensified display
Color pair 3 (yellow/black)	Intensified display
Color pair 4 (blue/black)	Intensified display
Color pair 5 (magenta/black)	Intensified display
Color pair 6 (cyan/black)	Intensified display
Color pair 7 (black/yellow)	Intensified display
Color pair 8 (white/black)	Intensified display

Function Key Labels

VPLUS simulates function key labels by "reserving" the last two lines of the screen. This reduces the available screen area to 22 lines.

Function key labels for IBM 3270 display stations appear similar to those for HP terminals. The label text appears on an IBM 3270 display station as intensified display on a blank background. The default labels on HP terminals and IBM 3270 display stations are identical, and cannot be configured by users.

Fields

You should not design a form that has contiguous fields, because the IBM 3270 display station requires an attribute byte preceding each field or text area. The attribute byte uses a physical screen location and appears as a blank space on the screen. The attribute byte indicates whether the field can be or has been changed.

You can create a space for both leading and trailing attribute bytes by inserting visible brackets before and after all fields. SNA DHCF can convert these visible brackets to invisible attribute bytes on IBM 3270s. For instance, if you defined the following three fields:

```
[08][05][52]
```

they would appear as shown below when the VPLUS form is displayed on an IBM 3270 screen.

```
*08**05**52*
```

The * represents an attribute byte. On the screen, the attribute byte appears as a blank.

If you use invisible brackets to delimit fields, you must provide a leading space before each field or text area. When the field or text begins in the first column of a line, insert the space at the end of the previous line. The "Home" position is an exception to this guideline; the attribute byte should be inserted in the bottom right position instead (reverse "Home").

Although the attribute byte is the most important concept regarding fields for IBM 3270 display stations, you should also be aware of the following information about fields.

Character Translation

As mentioned earlier, HP computer systems use ASCII character sets, whereas IBM computer systems use EBCDIC character sets. Consequently, SNA DHCF must use system default translation tables from the Native Language Subsystem (NLS) to translate characters from one system to the other.

If Native-3000 is the native language used, SNA DHCF translates all ASCII characters without alterations, except for the characters shown in table K-4. Note that the HP terminal bracket characters equate to text, not actual FORMSPEC brackets.

Table K-4. Differences Between HP and IBM Character Displays

HP Terminal Character	ASCII Hex Value	IBM 3270 Character	EBCDIC Hex Value
[	5B	¢	4A
]	5D	!	5A
!	21		4F
^	5E		5F

To display one of these characters on one system, you must specify the corresponding character on the other system. For instance, if you want to send an exclamation point (!) to an IBM 3270 user, your VPLUS application must specify this character as a right bracket (). If an IBM user wants to input a left bracket (), he or she must specify ¢.

Using FORMSPEC With DHCF

You can compile a forms file for an IBM 3270 and any HP terminal that supports VPLUS on MPE/V or MPE/XL by selecting the desired terminals from the Terminal/Language Selection Menu and ensuring that the forms file adheres to IBM 3270 requirements. You must compile a forms file for at least one HP terminal as well as an IBM 3270.

Certain FORMSPEC block-mode components prohibit you from running FORMSPEC interactively on an IBM 3270 display station while using SNA DHCF. However, you can use FORMSPEC in batch mode on an IBM 3270.

The following procedure provides a hypothetical example that uses FORMSPEC to modify a VPLUS form application for an SNA DHCF session.

1. Type the following command at the MPE prompt:

   ```
   RUN FORMSPEC.PUB.SYS
   ```

   FORMSPEC responds by displaying the Forms File menu.

2. Type the desired forms file name and press **Enter**. FORMSPEC responds by displaying the Main menu.

3. Type **T** at the Enter Selection field where the cursor is currently positioned. FORMSPEC responds by displaying the Terminal/Language Selection menu.

4. Type **X** beside the IBM 3270 field and at least one HP field as shown below. Even if your application is strictly for the IBM 3270, you must also select an HP terminal on this menu for diagnostic purposes. This enables you to display a form on an HP terminal if you are unable to display the form on an IBM 3270.

5. Press **Enter** after you have selected the desired terminals. If SNA DHCF is not installed on your system, FORMSPEC responds by displaying the message, `SNA DHCF not installed; necessary to compile for IBM 3270.` You must then either remove the IBM 3270 selection or make sure that SNA DHCF is installed on your system.

   If SNA DHCF is installed, FORMSPEC responds by displaying the Globals menu as shown below. Note that the Window Display Line shows the default of 24 lines.

6. Change the number of the Window Display Line field to either 0 (no window line), 1 (window line at top line of screen), or 22 (window line at line 22 above labels). If you inadvertently press **Enter** without changing this field, FORMPSEC responds by displaying the error message shown below.

   If this message appears, correct the error by specifying 0, 1, or 22 for the Window Display Line field and press **Enter**. FORMSPEC responds by displaying the Save Field menu.

7. Create or modify one or more forms. When you compile the form(s), you may encounter an error message. The following example shows one of the possible error messages for form "XYZ", which involves the attribute byte explained earlier in this appendix. (Refer to appendix B for a list of SNA DHCF-specific error messages.)

8. Go to form "XYZ" and press **Enter** to determine the nature of the error. FORMSPEC responds by displaying an explanatory message about the error and positions the cursor

at the error location as shown below.

In this example, the error is obvious. The field tag **D** must be moved one column to the right to line up with the other characters. Moving this field allows space for the attribute byte between the colon and the beginning of the field.

Using VPLUS Intrinsics With DHCF

From the group of all available VPLUS intrinsics, SNA DHCF does not support two of the intrinsics when they are run on IBM 3270 display stations. Six of the intrinsics can be used, but are altered because of IBM 3270-related differences. The remaining intrinsics can be used in the same manner as for an application to be run on HP terminals. The following sections discuss unsupported intrinsics and those that are affected by IBM 3270 differences.

Unsupported Intrinsics

If a VPLUS application calls an intrinsic unsupported for IBM 3270 display stations during an SNA DHCF session, you must modify the application before it can run successfully. Otherwise, the intrinsic may not function, or it may return an error message. The following intrinsics are not supported at this time.

`VPRINTFORM`	Generates an error when called. This intrinsic is dependent on an ASCII screen image, but the screen image on an IBM 3270 is EBCDIC.
`VPRINTSCREEN`	Generates an error when called.

In addition to these unsupported intrinsics, SNA DHCF does not support VPLUS internal routines called directly by an application.

Intrinsics Altered With SNA DHCF

The following intrinsics function differently for IBM 3270 display stations than for HP terminals.

`VCHANGEFIELD`	HP enhancement types map dynamically to IBM 3270 enhancement types.
`VOPENTERM`	The identifier is set to the IBM 3270 display station type (80; reserves 80-85 for 327x terminal types).
`VPUTWINDOW`	The message can contain up to 79 printable characters. Messages longer than this are truncated. Unprintable characters are removed.
`VREADFIELDS`	The *lastkey* parameter in the *comarea* contains the value that corresponds to the mapping of an IBM 3270 function key to an HP function key.
`VSETERR`	The message can contain up to 79 printable characters. Messages longer than this are truncated. Unprintable characters are removed.
`VSHOWFORM`	Uses *showcontrol* bits as for a non-caching, MDT, non-touch terminal. Enhancements are mapped to those available on an IBM 3270 display station. The following limitations apply: • The commands "Repeat/Append forms" do not roll a

partial form, because the terminal does not support scrolling and SNA DHCF does not simulate it. Instead, the commands roll up an entire form at a time.

- Brackets are not visible around fields.
- The function key area is simulated only in VPLUS mode.

SNA DHCF and VPLUS Utilities

Besides FORMSPEC, SNA DHCF affects certain other applications of the VPLUS-based forms management system as explained in the following sections.

ENTRY

ENTRY can run on an IBM 3270 if you have compiled the forms file for the IBM 3270. Initial prompts are in TTY-mode and the remaining prompts use standard VPLUS.

REFSPEC

REFSPEC cannot run on an IBM 3270 display station because it uses both VPLUS and file system intrinsics for terminal I/O.

REFORMAT

REFORMAT is a TTY-mode application and should run without modifications on an IBM 3270 display station.

HP32209B and HP32209S Utilities

The version identification utilities, HP32209B and HP32209S, are TTY-mode applications and should run without modifications on an IBM 3270 display station. The utilities check for SNA DHCF on MPE/XL systems. The following example shows the MPE response when you run utility HP32209B if SNA DHCF is installed on the system.

```
:RUN HP32209B.PUB.SYS

VPLUS INTRINSICS VERSION:   HP32209B.05.15

      ------- --------------
      SEGMENT    VERSION
      ------- --------------
        01    B'05'12'S01'00
        02    B'05'12'S02'00
        03    B'05'12'S03'00
        04    B'05'12'S04'00
        05    B'05'12'S05'00
        06    B'05'12'S06'00
        07    B'05'12'S07'00
        08    B'05'12'S08'00
        09    B'05'12'S09'00
        10    B'05'12'S10'00
        11    B'05'12'S11'00
        12    B'05'12'S12'00
        13    B'05'12'S13'00
        14    B'05'12'S14'00
      ------  ---------------
       DHCF     A0005005
END OF PROGRAM
:
```

The following example shows the MPE response when you run utility HP32209S if SNA DHCF/XL is installed on the system.

```
    :RUN HP32209S.PUB.SYS

    VPLUS INTRINSICS VERSION:   HP32209B.05.15

          ------- ---------------
          SEGMENT    VERSION
          ------- ---------------
            01    B'05'12'SO1'00
            02    B'05'12'SO2'00
            03    B'05'12'SO3'00
            04    B'05'12'SO4'00
            05    B'05'12'SO5'00
            06    B'05'12'SO6'00
            07    B'05'12'SO7'00
            08    B'05'12'SO8'00
            09    B'05'12'SO9'00
            10    B'05'12'S10'00
            11    B'05'12'S11'00
            12    B'05'12'S12'00
            13    B'05'12'S13'00
            14    B'05'12'S14'00
          -----   --------------


  VPLUS NATIVE MODE INTRINSICS VERSION: HP32209B.05.15

  FOR THIS USER:

     VPLUS NATIVE MODE IS ENABLED;
     VPLUS SWITCH TO COMPATIBILITY MODE IS DISABLED;


     WILL DEFAULT COMPATIBILITY MODE SL SEARCH TO
     SYSTEM SL


           DHCF     A0000001
    END OF PROGRAM
    :
```

L A Programmer's Guide to VPLUS

VPLUS is an HP tool used by programmers to design screens and implement an on-line application to collect and display information. With VPLUS, you are able to design screens which prompt the user for input in familiar business terms. VPLUS intrinsics are available to help you develop a customized application.

This note is intended for programmers who want to understand the purpose of each intrinsic and the sequence of calls. For each intrinsic, a discussion of the passed parameters, some programming tips, and some details of what the intrinsic does is presented. Also included is a summary of VPLUS enhancements which can make an existing or new application easier to use. A brief synopsis is presented on each enhancement. For more details on their use, please refer to the *HP Data Entry and Forms Management System (VPLUS/V) Reference Manual*. Note that in the remainder of this paper, this manual will be referred to as the *VPLUS/V Reference Manua*l.

This Application Note covers two topics to help you code your application.

- VPLUS Intrinsic Calls
- VPLUS Enhancements

This note assumes that the reader has some basic knowledge of the HP Data Entry and Forms Management System (VPLUS).

VPLUS Intrinsic Calls

ENTRY.PUB.SYS is an example of an application which uses VPLUS intrinsic calls. This program accepts data from a user and writes the information to an MPE file. The MPE file contains information in the user label for use by ENTRY.PUB.SYS.

Since ENTRY.PUB.SYS does not meet the needs of some business environments, many programmers develop customized programs to write the data to a database, an MPE file or a KSAM file; or to perform edits on the data. An understanding of the purpose of each intrinsic and the sequence in which the intrinsics are called is crucial to a successful program. The intrinsics can be logically grouped into one of six functions. An application program must use these functions in this sequence:

1. Opening Files
2. Preparing and Showing the Screen
3. Reading Data from the Screen
4. Editing
5. Returning Data to the Program
6. Closing Files

Illustration 1 in the appendix of this note is a flowchart of the VPLUS intrinsic calls. The flowchart illustrates the sequence of calls that an application program may follow. It groups the intrinsics into one of the six functions listed above. Note that it does not include examples of local form storage, softkey labeling, or interfacing with VPLUS batch file

Opening Files

Before a VPLUS application can run, it must open the necessary files. VOPENFORMF is called to open a formsfile. VOPENTERM is called to open a terminal. Note that VPLUS requires an HP block mode terminal. For a list of the supported terminals, refer to Appendix G of the *VPLUS/V Reference Manua*l.

VOPENFORMF

Purpose: to open the formsfile to be used by the application.

VOPENFORMF should be the first VPLUS intrinsic call executed in an application program. This intrinsic opens the formsfile specified in the call. Parameters passed to VOPENFORMF are the comarea array and the formsfile name. The comarea array is a 60 word communication area between the program and VPLUS. The comarea array contains assorted fields to be set before an intrinsic call.

One important field is CSTATUS or the status word. This is set on the completion of each VPLUS intrinsic call. The programmer should check this status after each intrinsic call to determine the success or failure of the call. If the status word is not zero, subsequent VPLUS intrinsic calls are not executed. If the program uses local forms storage or labeled softkeys, the FORM'STORE'SIZE and LABEL'OPTION words of the user's COMAREA must be set prior to calling VOPENFORMF. VOPENFORMF allocates additional space on the user's stack based on the values of these fields. Other words in the comarea array that need to be set prior to VOPENFORMF's call are:

- LANGUAGE which contains the language code of the calling program
- COMAREALEN which contains the length in words of the comarea array.

In addition, for BASIC/3000 programs, USRBUFLEN should specify the number of words needed for the comarea extension on the stack.

When VOPENFORMF is executed, it opens the formsfile and allocates space on the stack for the comarea extension. The comarea extension is used by VPLUS to protect data between calls, to provide a buffering mechanism so that terminal reads are transparent to the user, and to aid in directory searches for formsfile information. Illustration 2 in the appendix of this note contains a layout of the tables which are in the comarea extension. For most programming languages, the comarea extension resides in the DL-DB area of the stack. For BASIC/3000, the comarea extension is declared as a variable of the program and resides with the global variables.

VOPENTERM

Purpose: to open the terminal as a file and set the terminal for block mode.

Another intrinsic which performs an open function within VPLUS is VOPENTERM. This intrinsic is called after VOPENFORMF. VOPENTERM does exactly what its name suggests. It opens the terminal as a file. In addition it sets the terminal in block mode and initializes the user function keys to contain the escape sequences which VPLUS expects.

Two parameters are passed to VOPENTERM: COMAREA and TERMFILE. COMAREA is the comarea array which VPLUS uses to communicate with the program. The words within the comarea array that VOPENTERM requires before the call are: CSTATUS, LANGUAGE, COMAREALEN and FORM'STORE'SIZE. These words were set at the time VOPENFORMF was called and therefore do not need to be reinitialized. Upon the completion of the call, VOPENTERM returns additional information (such as CSTATUS) in the comarea array which indicates the success or failure of a call. It also returns FILERRNUM which is the file system error if it could not open the terminal; FILEN which is the MPE file number of the terminal file; IDENTIFIER which contains a unique terminal ID assigned to a specific terminal; and ENVIRON which contains the logical device number of the terminal.

TERMFILE specifies a file name which is opened. This file defaults to the logical device from which the program is being run.

When debugging your VPLUS application, the messages displayed from your programs are only displayed within the unprotected fields on a form. For debugging purposes, it is possible to redirect the screen to another logical device so that an error message sent to $STDLIST is displayed in its entirety.

To redirect the VPLUS screen to another LDEV, you should:

1. Ensure that the other terminal's LDEV number is AVAIL by issuing a :SHOWDEV command. The terminal must be logged off. Do not use the keyboard or the terminal becomes unavailable at that time. The terminal must be configured at the same baud rate as the port of garbage will print when VPLUS displays the screen. This is because no speed sensing is done when the screen is routed to the alternate terminal.

2. Issue a file equation as follows:

   ```
   :FILE {termfilename};DEV=xx
   ```

   {termfilename} is the name used in your call to VOPENTERM; 'xx' is the LDEV of the terminal to which you are redirecting the screens

Another method that can be used in your application is to redirect the display of the error message from the program to another device. This allows the screen to be displayed on the terminal from which you are running the program, but the error messages will be directed elsewhere. This method of redirecting $STDLIST is the easiest to use.

1. Issue a file equation to the device where you would like the displays directed, for example:

   ```
   :FILE TEST;DEV=LP
   ```

   or

   ```
   :BUILD TEST;REC=-132,1,F,ASCII;DEV=DISC :FILE TEST,OLD;DEV=DISC
   ```

   or

   ```
   :FILE TEST;DEV={LDEV # of terminal, which must be AVAIL
   ```

2. After setting the file equation in step 1, run the program as follows:

   ```
   :RUN {vprog};STDLIST=*TEST
   ```

Preparing and Showing the Screen

The next stage in a VPLUS application program is to prepare and show the screen. Preparing the screen to be displayed involves retrieving the form definition and initializing fields with the desired values. Optionally, it may include initializing the terminal window with a message and labeling the function keys. The intrinsics which prepare the screen to be shown are VGETNEXTFORM which retrieves the specified next form's design; VINITFORM and/or VPUTBUFFER which places values into fields on the screen; VPUTWINDOW which places a message to be shown on the terminal window; and VSETKEYLABEL(S) which can label function keys. After all the information to be displayed is set up, VSHOWFORM is called to display the form.

This section discusses the VGETNEXTFORM, VINITFORM, VPUTBUFFER and VSHOWFORM intrinsics. When used in a program, the intrinsics are called in the order listed.

VGETNEXTFORM

Purpose: to retrieve the next form's definition.

VGETNEXTFORM is the next intrinsic called after VOPENTERM. It is the first intrinsic to prepare and show the screen. It may also be called when a new form is to be displayed on the terminal.

The only parameter passed to VGETNEXTFORM is COMAREA. Within the comarea array, there are several variables pertinent to VGETNEXTFORM such as NFNAME, REPEATAPP, FREEXAPP and CFNAME. NFNAME is the variable containing the next form's name. CFNAME is the variable containing the current form's name. After the call to VGETNEXTFORM, the CFNAME variable in the comarea array is updated to contain the name of the form just read from the formsfile.

REPEATAP is the repeat append option and will take on a numeric value from 0 - 2. These values are defined as follows:

0	Do not repeat the current form 1
1	Current form is to be repeated
2	Current form is to be repeated and appended

FREEZAPP is the freeze append option. This will also take on a numeric value from 0-2. The values for FREEZAPP follow:

0	Clear the current form before displaying the next form
1	Next form is to be appended to the current form
2	Current form is to be frozen before the next form is appended

VGETNEXTFORM retrieves the next form based on the value of two words in the COMAREA: REPEATAPP and NFNAME. If the repeat append word indicates the current form is to be repeated (REPEATAPP=1) or the current form is to be repeated and appended to itself (REPEATAPP=2), VGETNEXTFORM does not get the next form. REPEATAPP must be set to 0 before VGETNEXTFORM will retrieve the next form. The next form is determined by NFNAME. If NFNAME is blank, it gets the $HEAD form.

Other options for NFNAME, other than an actual form name are:

```
$END
$HEAD
$REFRESH
$RETURN
```

Upon the completion of the call to VGETNEXTFORM with $END as the NFNAME, the CFNAME becomes $END. The application must check the CFNAME to determine to shut down the application.

With $HEAD, the head form which is specified in FORMSPEC, is retrieved. Within FORMSPEC, a specific form may be designated as the head form. If this field is left blank, the head form becomes the first form in the formsfile (alphabetic order of forms).

When VGETNEXTFORM is called with $REFRESH, the terminal is reinitialized. If there are forms in local forms storage, the forms are cleared. Only the current form is displayed with $REFRESH, therefore prior appended screens are lost. Data, screen design, windows and softkeys are displayed again. An alternative to using $REFRESH which takes less overhead is to simply display the data again. In many cases, where the data is overlaid but the screen design is intact, this method is sufficient. An example of when this type of refresh can be used is when another user has sent a WARN message through the user's form. This refresh method would simply force data and field enhancements to be rewritten to the screen.

An example of the alternate method is:

```
VINITFORM (COMAREA)
SHOWCONTROL (14:1)=1
```

[COBOL programmers must add 2 to SHOWCONTROL]

```
VSHOWFORM (COMAREA)
SHOWCONTROL (14:1)=0
```

[COBOL programmers must subtract 2 from SHOWCONTROL]

Lastly, VGETNEXTFORM may be called with $RETURN. Before performing VGETNEXTFORM, REPEATAPP must be 0 as in all the above cases. VPLUS keeps track of the forms most recently used by your program. $RETURN returns the last form called. If the current form is $HEAD, it returns $HEAD again (no forms file access is actually done). The form file definition which is retrieved by VGETNEXTFORM is stored in an area of the user's stack called the comarea extension.

VINITFORM

Purpose: to initialize fields on the current form.

VINITFORM is called after VGETNEXTFORM in order to initialize any fields on a form. For example, on the field menu, the designer may specify an initial value. Or within the field processing section of the field menu, a designer may specify INIT phase specifications such as setting the field to a value of another field on the form.

VINITFORM uses any initial value specified in the FIELD MENU and performs any processing specifications defined in the INIT phase.

Any field that does not have a specified initial value is by default set to $EMPTY (all blanks). If this is a child form, the values for any fields that do not have an initial value are retrieved from the parent form. The only parameter passed to VINITFORM is COMAREA. The fields within the comarea array which VINITFORM may set are listed in the VPLUS/V Reference Manual.

Any initialized values are placed in the comarea extension's data buffer.

VPUTBUFFER

Purpose: to place data from a program buffer into the VPLUS data area.

Before the form is displayed with the initialized fields, additional information may be retrieved from a data base, KSAM file, MPE file, or from within the program. This is application dependent. Once the data is retrieved with a DBGET or FREAD call, VPUTBUFFER transfers the data from the program's form buffer to the VPLUS data buffer. VPUTBUFFER, however, does not format the data being sent to the data buffer (e.g., strip leading zeros or format signs on numeric fields); this must be handled by the application before VPUTBUFFER is called. The comarea extension contains its own copy of the data buffer which aids VPLUS to interface with the terminal.

VPUTBUFFER is an optional step for preparing and showing the screen function. It is used if information from other sources needs to be displayed on the screen. An example is showing an employee's name which is retrieved from a personnel data base after the user has entered the employee's number.

The call to VPUTBUFFER in pseudocode looks like:

```
  VPUTBUFFER (COMAREA, BUFFER, BUFLEN)
```

There are three parameters passed to VPUTBUFFER: COMAREA, buffer, and buffer length. The

COMAREA words have been set from preceding VPLUS intrinsic calls. If CSTATUS word of the user's COMAREA is not zero, the call to VPUTBUFFER does not execute. The buffer is the name of the program's form buffer whose contents are to be put to the form. The third parameter is the length of the buffer in bytes.

VSHOWFORM

Purpose:	to display screen definition, data, window function keys, enhancements or terminal screen.

This intrinsic takes the current form as defined in the comarea extension form definition and displays it on the screen. A prior call to VGETNEXTFORM places the form definition in the comarea extension. VSHOWFORM displays the field values from the data buffer of the comarea extension. The content of the data buffer is affected by calls to VINITFORM, VPUTBUFFER or VREADFIELDS. VSHOWFORM also uses the global/form function keys and window buffer from the comarea extension to display the softkey labels and the window message.

The only parameter which is passed to VSHOWFORM is COMAREA. The SHOWCONTROL word in the COMAREA allows the programmer control of several VSHOWFORM options. The following paragraphs discuss some of the options and how they are used.

Bit 15=1	Forces the form to be displayed again. This overrides the VPLUS default. This may be useful when using process handling and returning to the father process' screen. In some cases, VPLUS does not show the father process screen when control is returned to the father process because it believes the father screen is the current form. Bit 15 must be set to 1 in this case.
Bit 14=1	Forces the window to be displayed again. The other intrinsics which force the window to be displayed again are VPUTWINDOW, VSETERROR, VGETNEXTFORM with NFNAME=$REFRESH.
Bit 13=1	Forces the data to be displayed again. VPLUS normally does not re-display values that are already on the screen. Setting bit 13 to 1 overrides the default, and may be used as another alternative to $REFRESH. In most cases, forcing the data to be displayed again is sufficient rather than using $REFRESH which will re-display data, screen design, window and softkeys. (Refer to the example of how this is performed in the preceding VGETNEXTFORM discussion).
Bit 10=0	Enables the keyboard (default).
Bit 10=1	Disables the keyboard.VREADFIELDS locks the keyboard so that no user input can take place until the screen is displayed. VSHOWFORM unlocks the keyboard so that a user may enter data after the screen is displayed. If more than one VSHOWFORM is called in a row without an intervening call to VREADFIELDS, bit 10 should be set to 1. After the last VSHOWFORM is called, bit 10 should be set back to 0 to allow user input of data.
Bit 9=0	Do not preload current form into local form storage (default).
Bit 9=1	Preload form into local form storage.

For terminals that have local form storage, the current form may optionally be loaded into local form storage if it is not already present.

Bit 8=0	Do not enable A/M/U (default).
Bit 8=1	Enables A/M/U.

If bit 8 is set to 1, users may access the A/M/U. Currently on VPLUS version B.04.20, a VENVCNTL.PUB.SYS file can also control this option; the format of this file is discussed in the *Communicator*, Volume 2, Issue 13 (UB-DELTA-1).

Bit 0=0	Do not enable the touch screen feature (default).
Bit 0=1	Enable the touch screen feature.

As of VPLUS version B.04.10, VREADFIELDS can be used to read a field number that has been "touched". Terminals that support the touch screen feature are listed in the *VPLUS/V Reference Manual* Appendix G.

Reading Data from the Screen

After displaying the screen along with any initial values, the user is ready to input data. VREADFIELDS triggers a read from the user. The read is satisfied when the user presses E or one of the function keys. Also if the HPTOUCH feature is implemented, the read is satisfied when a user selects a field on the form. For more information on the HPTOUCH feature, refer to the *VPLUS/V Reference Manual*. The VPLUS Enhancement section briefly discusses this feature.

VREADFIELDS

Purpose: to trigger a read from the user.

VREADFIELDS allows users to enter data. The read is satisfied if the user presses E or any other function key. If E is pressed, data is transferred from the screen to the data buffer in the comarea extension. Data is read in screen order, from top to bottom, and left to right.

If a function key is pressed, control is returned to the program without any data being transferred to the VPLUS data buffer in the comarea extension. Instead, the LASTKEY word in the comarea array is set. LASTKEY contains a value corresponding to the function key that satisfied the read trigger. LASTKEY will take on a value as follows:

0	Enter Key
1 - 8	Corresponding to F1-F8

If the HPTOUCH feature is implemented, VREADFIELDS is satisfied when a user selects a field. IN this case, no data is transferred to VPLUS's data buffer. Instead, the LASTKEY word in the comarea array is set to a negative number which indicates the field number selected. The number is negative to distinguish it from the function key's numbers which are positive 1 through 8.

The only parameter passed to VREADFIELDS is COMAREA. Within the COMAREA, the TERMOPTIONS word can be set to perform some advanced functions. These include setting up automatic reads or having a timed read.

AUTOREAD is used when VREADFIELDS is terminated by a function key instead of E. For example, if an order processing clerk is entering order information on a screen, then presses a function key labeled “MODIFY”, the program needs to read the data on the screen. Without the AUTOREAD option set, the program would not read the screen. The program needs to interrogate the LASTKEY word of the COMAREA for an i value. If a non-zero value is found, it needs to proceed with enabling AUTOREAD.

AUTOREAD is enabled by setting bit 14 of the TERMOPTIONS word to 1. For COBOL programmers, ADD and SUBTRACT may be used to set bit 14. For example, in pseudocode:

```
VREADFIELDS (COMAREA)
    IF LASTKEY = 1 THEN
    TERMOPTIONS(14:1)=1
```

[COBOL programmers must add 2 to TERMOPTIONS]

```
    VREADFIELDS(COMAREA)
    TERMOPTIONS(14:1)=0
```

[COBOL programmers must subtract 2 from TERMOPTIONS]

```
ENDIF
```

Another feature which is enabled through the TERMOPTIONS word of the comarea array is setting a timed read. The TERMOPTIONS word bit (10:1) enables a times read. Another word in the comarea array, USER'TIME, defines the time limit in seconds. Again, COBOL programmers must use the ADD and SUBTRACT to set bit 10. For example, to enable a timed read for 60 seconds in pseudocode:

```
USER'TIME = 60
TERMOPTIONS(10:1)=1
```

[COBOL programmers must add 32 to TERMOPTIONS].

```
VREADFIELDS(COMAREA)
IF COM'STATUS = 160 THEN
PERFORM TIME-OUT-PROCEDURE
ENDIF
TERMOPTIONS(10:1)=0
```

[COBOL programmers must subtract 32 from TERMOPTIONS].

Editing

Once the data is read from the screen into VPLUS's data buffer, edits may be performed. The edits may be performed in as many as three passes. On the first pass, the editing process checks that all required fields are present, the data input by the user is consistent with the data type defined for the field, and the data meets rules set out by the user (for example, matching a pattern or being within range of values). Optionally, a second editing pass may be performed by the program. Edits can be designed into the program. An example of a programmatic edit is verifying that an employee number exists in the personnel data base before allowing a modify of an employee record. Optionally, a third pass of editing may be performed. The third pass of editing performs any totaling or formatting functions. For example, if a form contains hours worked for each day of the week, a first level editing pass would verify that the hours input for each day is less than 24 hours. In this example, we will assume that there are no programmatic edits in the second pass of editing. However, in the third pass of editing, a weekly total is accumulated and the numbers are right justified.

This section on "Editing" will cover the three phases of editing: VFIELDEDITS, optional programmatic edits, and VFINISHFORM.

VFIELDEDITS

Purpose: to verify the data entered.

This intrinsic is called to verify the data entered. If the field is defined as OPTIONAL and is left blank, no further processing is performed. To perform processing on a blank field (for instance, to set it to a value), the field type should be defined as PROCESS. VFIELDEDITS performs the following checks if the field is a REQUIRED field type or a PROCESS field type and the field contains data.

- Field type - If the field type is REQUIRED and no data is present, the field is flagged in error.
- Data type - If the data type is invalid, the field is flagged in error. The data input by the user must match the data type defined for the field.
- Field processing specification statements - The user may define processing specification statements within each field menu. If the data does not meet the specifications set up by the user, the field is flagged in error.

If VFIELDEDITS detects any errors on the checks listed above, the field is set in error and the NUMERRS word of the user's COMAREA is incremented by one. The only parameter passed to VFIELDEDITS is COMAREA. When the intrinsic completes, it may set the NUMERRS word within the comarea array. NUMERRS contains a count of the number of fields on the form detected in error. Note that a CSTATUS of 0 only indicates that the intrinsic executed properly, not that no errors were found; this information is contained in NUMERRS.

In looking at Illustration 1, if there are any errors (NUMERRS > 0) the program calls an error processing routine. The error processing routine calls VERRMSG to retrieve the appropriate error message of the first field in error. The error processing routine then calls VPUTWINDOW which puts the error message to a window buffer maintained by VPLUS. It next calls VSHOWFORM which highlights all fields in error and displays the error message in the terminal window. VREADFIELDS is called to prompt the user to input the correct data. Then VFIELDEDITS is called once again. This loop is performed until there are no errors flagged by VFIELDEDITS.

Optional Programmatic Edits against Data

After passing the first phase of editing, additional checks may be desired. For example, a programmatic check could be made to verify that the employee number entered exists in the company's data base. Since this type of check cannot be done in FORMSPEC, it needs to be implemented by the program with a call to a data base, KSAM or MPE file.

Before any user editing is performed, the program needs to retrieve VPLUS's data buffer into its own program buffer. A call to VGETBUFFER or VGETFIELD or VGETtype returns the data to the program's buffer. At this point, the data can be verified against a database or KSAM file or MPE file. If the programmatic edits detect an error, VSETERROR is called to set the field in error.

VSETERROR is another VPLUS intrinsic available to the programmer so that when programmatic edits detect invalid data, the field may be highlighted and set in error. The pseudocode for VSETERROR looks like:

```
VSETERROR (COMAREA, FIELDNUM, MESSAGE, MSGLEN)
```

VSETERROR sets the field in error if it is not already in error and increments the NUMERRS word of the user's COMAREA by one. After the call to VSETERROR, NUMERRS reflects the number of fields detected in error on the form. The second parameter of VSETERROR is the field number to be set in error. The third and fourth parameters of VSETERROR are the customized error message and the error message length, respectively. If this is the first field in error, the customized error message appears in the terminal window on a call to VSHOWFORM.

In looking at Illustration 1, if the optional programmatic edits detect an error, the field is set in error by VSETERROR. The subsequent VSHOWFORM displays the programmer's custom error message and highlights the fields in error. VREADFIELDS is then called after VSHOWFORM to allow the user to input the corrected data. The data is next taken through all the passes of editing once again - beginning with VFIELDEDITS, next with the programmatic edits, and finally with VFINISHFORM.

VFINISHFORM

Purpose: to perform final editing.

Once the validity of data has been established with VFIELDEDITS and it has passed optional programmatic editing, VFINISHFORM can be called to perform the final totaling and editing processes. VFINISHFORM executes any processing statements defined in the FINISH phase. If there are no FINISH phase statements in the form, it is not necessary to call VFINISHFORM.

FINISH phase statements are input through FORMSPEC in the field menu at the field processing specification section. Formatting and totaling are typical functions performed in this phase. An example of the formatting at FINISH phase is right justifying a number, or stripping out a decimal point or filling a numeric field with leading zeros.

The only parameter passed to VFIELDEDITS is COMAREA. Within the comarea array, VEDITFIELDS may set the NUMERRS word. If there are any field type, data type or editing errors, the field is set in error and NUMERRS is incremented.

In looking at the flowchart, if there are any errors (NUMERRS>0), VERRMSG retrieves the appropriate error message. VPUTWINDOW puts the error message in VPLUS's window buffer. When VSHOWFORM is called, all fields in error are highlighted and the window displays the error message retrieved by VERRMSG. VREADFIELDS allows the user to input corrected data. Again, the flowchart passes through all levels of error checking from VFIELDEDITS through VFINISHFORM. Once the data passes all levels, the program proceeds to the next step.

Returning Data to the Program

After the data is verified by VFIELDEDITS, optional programmatic edits, and VFINISHFORM, it is returned to the program's form buffer with a call to VGETBUFFER. There are other intrinsics to return from VPLUS's data buffer to the program buffer such as VGETtype and VGETFIELD. These two intrinsics can be used to retrieve one field's worth of data to a program buffer. Refer to the *VPLUS/V Reference Manual* on their use.

This section covers the VGETBUFFER intrinsic.

VGETBUFFER

Purpose: to return values from VPLUS data buffer to the user's program buffer.

When VREADFIELDS is performed, it returns user input data to the VPLUS data buffer. When VFIELDEDITS and VFINISHFORM are performed, the editing and formatting is performed on VPLUS's copy of the data buffer. In order to return the data to the program buffer, a call to VGETBUFFER is necessary.

The call to VGETBUFFER in pseudocode looks like:

```
VGETBUFFER (COMAREA, BUFFER, BUFLEN)
```

The number of bytes that VGETBUFFER returns into the program form buffer is determined by BUFLEN (the third parameter of the call) and DBUFLEN (word in comarea array which contains the buffer length in bytes of the current form). DBUFLEN is set by the call to VGETNEXTFORM which retrieved the current form. VGETBUFFER transfers the lesser number of bytes between BUFLEN and DBUFLEN into the program buffer. VGETBUFFER returns all the data from the screen in a left-to-right and top-to-bottom order. Display-only fields are also returned into the program buffer and therefore you will need to ensure that the program buffer is set up to accommodate this data.

There are also two other routines that transfer data from the VPLUS data buffer into the program buffer.

- VGETFIELD - transfers one field from the VPLUS data buffer to a character buffer.
- VGET{type} - transfers one field from the VPLUS data buffer to a numeric defined field, such as an integer, double integer, real, or long.

VGETFIELD and VGETtype are useful to you if you want to return one field at a time to your program buffer. If the form design is changed so that fields are moved around within the form, VGETBUFFER's call requires that your program buffer matches the changes you have made to the screen. With VGETFIELD and VGETtype calls, the program buffer's definition is independent of the order in which fields are organized on the screen.

Once the data resides in the program buffer, the program may write the data to a database, KSAM or MPE file.

Closing Files

Once data collection has finished, the terminal and formsfile may be closed. Signaling the end of data collection can be accomplished several ways and is application dependent. Some programmers use 8 to signal an exit. When VREADFIELDS has finished and LASTKEY is equal to 8, the program performs the closing routines.

VCLOSETERM

Purpose: to close the terminal file and reset the terminal out of block mode.

This takes the terminal out of block mode and resets the straps. VCLOSETERM only partially restores the terminal's configuration. One of the things that it does not do is to restore user function key labels.

If a VPLUS application terminates abnormally, VCLOSETERM is not done. Therefore, the terminal is not taken out of block mode and echo is not turned back on. Generally, a hard reset and an "e :" can reset the terminal back to a usable state.

VCLOSEFORMF

Purpose: to close the formsfile.

In all languages except BASIC/3000, VCLOSEFORMF releases stack space allocated by the VOPENFORMF intrinsic. The stack space released was for the comarea extension. In BASIC/3000, since the comarea extension is declared as a global variable, the space is not released with a call to VCLOSEFORMF.

In addition, VCLOSEFORMF performs an FCLOSE on the formsfile.

VPLUS Enhancements

There have been many changes and enhancements to VPLUS since version B.04.10. This section will briefly discuss some of the changes and enhancements and how they can be of use to the programmer. The ones reviewed in this section are:

- HPTOUCH Support
- Cursor Positioning
- 264X Function Key Labels
- Enhanced VGETFORMINFO
- Batch Mode Formspec Enhancements
- Color Support for 2627A and 2392A Terminals
- VCHANGEFIELD
- VPLUS Environment Control File

HPTOUCH Support (Introduced on VPLUS B.04.10 in MPE G.01.01)

With HPTOUCH support in VPLUS, a programmer may design a form that is to be used with the HP150, the 2393A or the 2397A. Users may select a field by touching the field on the screen. When a field is selected, VREADFIELDS interprets the row and column coordinates into a field number.

VREADFIELDS returns the field number as a negative number to LASTKEY. Negative numbers are returned into LASTKEY so as to distinguish from the positive number that is returned when a function key is hit. LASTKEY is set to -999 if the area touched is not a field. HPTOUCH is enabled by setting bit 0 of the SHOWCONTROL word in the user's COMAREA to 1. An example of coding the HPTOUCH feature is outlined in the *VPLUS/V Reference Manua*l in Appendix E. Currently, HPTOUCH is supported on the 2393A, 2397A and HP150 terminals. For an up-to-date list of the terminals supporting HPTOUCH, refer to Appendix G of the *VPLUS/V Reference Manua*l.

Cursor Positioning (Introduced on VPLUS B.04.10 in MPE G.01.01)

VPLUS displays the screen and positions the cursor to the first field on the form. With some applications, positioning the cursor elsewhere would eliminate the requirement that the user press T to get to the desired field. VPLACECURSOR is a new intrinsic introduced in VPLUS version B.04.10 which allows the cursor to be positioned at an unprotected field. An unprotected field is any field in which the user is allowed to enter data. Protected fields may be headings or display-only fields. If the cursor is positioned at a display-only field, an error is returned. VSHOWFORM should be called first before positioning the cursor. The parameters of VPLACECURSOR are:

- COMAREA
- FIELDNUM - If the field number parameter is negative, it implies screen order number. If the field number parameter is positive, field number creation order is implied. The

screen order number is a relative number of the field on the form. For example, if there are five fields on a form and the programmer wants to place the cursor on the fourth field, the FIELDNUM parameter would contain a "-4". On the other hand, if the FIELDNUM parameter is positive, it implies field number creation order. This number is assigned to the field when the field is created through FORMSPEC. This number stays with the field unless the form is renumbered through batch mode FORMSPEC or if the field is deleted from the form

264X Function Key Labels (Introduced on VPLUS B.04.10 in MPE G.01.01)

With 264X terminals, there are no function key labels like the 262X or 239X terminals. As of VPLUS version B.04.10 which was introduced in MPE version G.01.01, function key labels may now be displayed on 264X terminals. With this version of VPLUS, function key labels appear in lines 23 and 24 of the terminal display on a 264X terminal. With 264X terminals, function keys may be displayed by performing the following steps:

1. Select "Y" instead of "X" in the 264X box of the TERMINAL/ LANGUAGE SELECTION MENU.
2. Define the labels using the same methods as with other HP Terminals, for example, enter the labels using the FORM FUNCTION KEY LABELS MENU, or enter the labels using the GLOBAL FUNCTION KEY LABELS MENU, or set the labels in your program calls to VSETKEYLABEL or VSETKEYLABELS.
3. Set the LABELOPTION word in the COMAREA to 1 before a call to VOPENFORMF (as with all other terminals).

VGETFORMINFO Enhancement (Enhanced on VPLUS B.04.10 in MPE G.01.01)

Prior to this enhancement, if VFIELDEDITS detected a field in error, the total number of errors would be set in the NUMERRS word of the user's COMAREA. All fields would be highlighted upon a call to VSHOWFORM. No information regarding which fields were found to be in error is passed to the program.

With VPLUS version B.04.10, VGETFORMINFO has been enhanced to return information on all fields in error. The information may be used by the program if a special error routine is to be performed when a certain field is in error. The information buffer returned by VGETFORMINFO has been expanded to include a 16-word array. This 16-word array is a bit map of all the fields on the form and can accommodate the maximum of 256 field numbers. Field numbers may range from 1 to 256, although a form may have a maximum of 128 fields on the form. If the bit corresponding to the field is set to 1, the field is in error.

To decode and set the bits, a new intrinsic called BITMAPCNV is available and documented in the *VPLUS/V Reference Manual*, Appendix I.

Batch Mode FORMSPEC (Enhanced on VPLUS B.04.10 in MPE G.01.01)

There have been several enhancements to batch mode FORMSPEC. Chapter 7 of the

VPLUS/V Reference Manual describes how to use batch mode FORMSPEC. Three new commands are available to the programmer:

- FIELD - This command allows, several field attributes to be updated in batch mode. Any of the following can be changed: field name, field enhancements, field type, data type, initial value.
- RENUMBER - This command reassigns field numbers to their screen order numbers. This is particularly useful for forms which have deleted fields and are approaching the maximum field number (256).
- FKLABELS - This command creates or updates a form's function key labels.

To further enhance batch mode compiling in FORMSPEC, two new JCWs have been introduced: FORMSPECERRORJCW which indicates the count of the number of errors during a compile and FORMSPECWARNJCW which indicates the count of the number of warnings during a compile.

Color Support for 2627A and 2397A (Introduced on VPLUS B.04.15 in MPE G.01.04)

Starting with VPLUS version B.04.15 which was introduced on MPE version G.01.04, 2627A and 2397A color terminal users may have forms displayed with specific colors defined for a field. VPLUS has defined eight color pairs which can be used to enhance fields, highlight fields in error, and display the window. Digits 1 through i correspond to the default color pairs. The color pairs are defined in Chapter 3 of the *VPLUS/V Reference Manual*. These color pairs can be used along with the normal enhancement set of H, I, B, U, S or NONE for field, error or window enhancements. To use the color enhancements, the formsfile must have an "X" in the HP2627 box in the TERMINAL/LANGUAGE SELECTION MENU in FORMSPEC.

VCHANGEFIELD (Introduced on VPLUS B.04.15 in MPE G.01.04)

Prior to VPLUS version B.04.15, a field defined through FORMSPEC could not be changed in the program. Therefore, if a programmer decided that a field defined as a "REQUIRED" field type should now be considered a "DISPLAY" field type, this could not be done. With VCHANGEFIELD, this is now possible. This intrinsic dynamically alters a field on a form. The changes are temporary, and are not posted to the formsfile. Some of these changes are:

- Changing field enhancements to the following: H, I, B, U, NONE. Changing field enhancements to S (security) is not available. As of version B.04.20, VCHANGEFIELD has been expanded to allow field enhancements to include a color pair in addition to H, I, B, U or NONE.
- Changing field types to any of the following: Optional (O), Display (D), Processing (P), Required (R).
- Changing data type to any of the following: CHAR, DIG, NUMn, IMPn, DMY, MDY, YMD.

User Environment Control File (Introduced on VPLUS B.04.20 in MPE G.02.01)

With VPLUS version B.04.20 introduced in MPE G.02.01, there is an MPE file called `VENVCNTL.PUB.SYS` which is opened and read on the `VOPENTERM` call. Currently, the VPLUS USER ENVIRONMENT CONTROL file can control two items when `VOPENTERM` is executed. Two options have been implemented for the user environment control file.

- The first option is an abbreviated terminal query. When `VOPENTERM` is called, it goes through an involved process to identify the terminal type. With the VPLUS user environment control file, this terminal identification process can be shortened. This option should only be used if it is known that a query will return a valid terminal ID. To effect a shortened terminal query, place a “1” in column 1 of the `VENVCNTL` file.
- The second option is the ability to enable A/M/U without using a program. Instead of setting the `SHOWCONTROL` word of the user's `COMAREA` in your program, use the `VENVCNTL` file. Placing a “1” in column 2 of the `VENVCNTL` file enables the keys. This option is particularly useful if you do not have source code for the software packages that you are running, but would like to enable A/M/U.

To implement a user environment control file for VPLUS, build the file:

```
:BUILD VENVCNTL.PUB.SYS;DEV=DISC;REC=-80,1,F,ASCII
```

Use any type of editor tool to place “1” in column 1 and/or column 2. When `VOPENTERM` executes, it looks for the user environment control file. If it doesn't find it, VPLUS operates as usual. If the file exists and columns 1 and/or 2 are set, VPLUS will perform the abbreviated terminal query and/or the enabling of keys, depending on what columns are set.

It is also possible to have a `VENVCNTL` file in a different group and account than `PUB.SYS`. If the file is in another group and account, this file equation is necessary:

```
:FILE VENVCNTL.PUB.SYS=VENVCNTL.mygroup.myaccount
```

For more information on how to implement the features, refer to the *VPLUS/V Reference Manual*.

VPLUS & Multiplexers

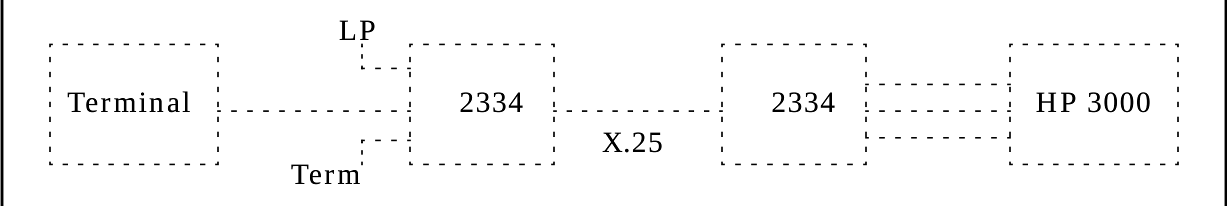
Using VPLUS on a Pad-Terminal(Connected to a Cluster Controller HP 2334A)

What is a CLUSTER CONTROLLER?

When you use a Packet Switching Network (PSN) for data communication, you need equipment to do the packing and unpacking of the data at both ends. A CLUSTER normally will receive the packets for different users, unpack them and change the transmission mode from synchronous to asynchronous. This way you can connect several different asynchronous devices, such as printers and terminals to one X.25 packet line (on a HP 2334A up to 16).

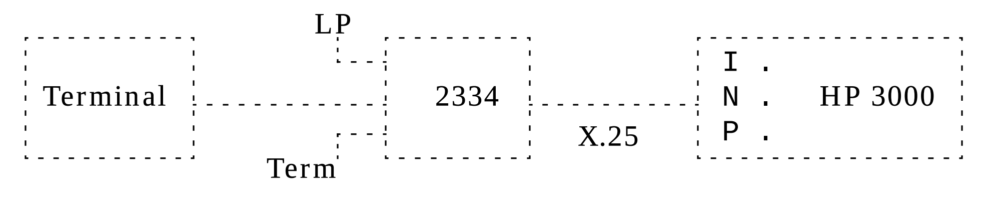
At the computer side you can use another CLUSTER and connect the asynchronous ports to the ATP or ADCC ports of the HP 3000. This kind of connection is called STATISTICAL MULTIPLEXER.

Figure L-1. Statistical Multiplexer



The other way is to connect the X.25 packet line directly to an INP-card in the HP 3000 computer. To support the INP, you need special DS- or NS-software. The software and the INP-card will do the packing and unpacking of the data received. This configuration is the CLUSTER CONNECTION and we call it Packet Assembler / Dissassembler (PAD).

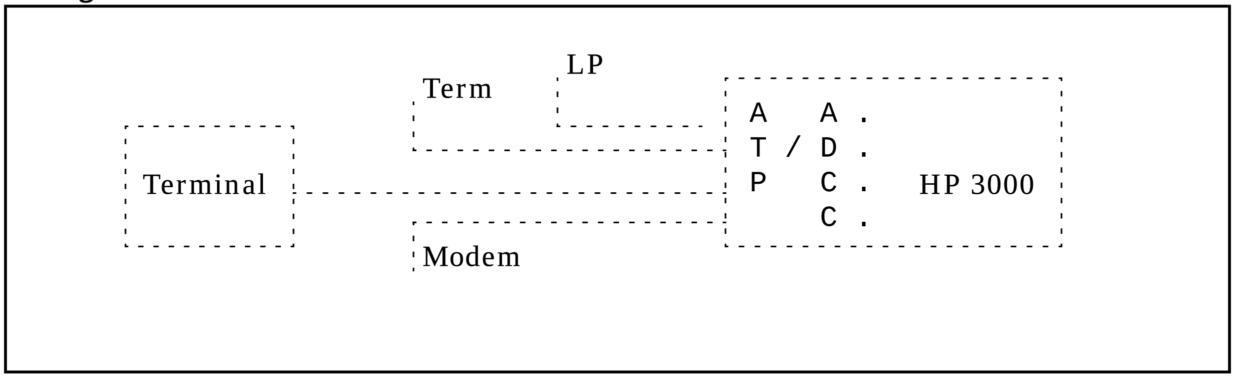
Figure L-2. PAD or CLUSTER Connection to an HP 3000 with an INP



First we will describe the minor differences between using the terminal directly to an ADCC / ATP port on an HP 3000, or to a port on a PAD (HP 2334A).

1. Terminal configuration when used to a direct line (ADCC / ATP):

Figure L-3. Direct Connection to an ATP- or ADCC-Port



```
DATACOMM :

  BaudRate : line speed (9600, 4800, 2400, 1200, ..)
  Parity/Bits: Par/# of bits (None/8, 0'S/7, ODD/7, EVEN/7)
  Enq/Ack : YES use enq/ack handshake
  Asterisk : OFF no indicator is shown
  Chk Parity : NO we don't check for parity errors
  SR(CH) : LO we don't select any Baudrate
  CS(CB)Xmit : NO we don't use modem signals
  RecvPace : NONE no XON and XOFF handshake is done
  XmitPace : NONE on receive or transmit
```

NOTE Do not turn RecvPace or XmitPace to on at a 2392A terminal if it is connected directly to an HP 3000. An XOFF (ctl S) could hang the terminal. (Only POWER-OFF the terminal will reset this situation).

```
TERMINAL :

Datacomm/ExtDev : PORT1/PORT2
Keyboard : USASCII (or other keyboards)
LocalEcho : OFF the HP 3000 will echo
CapsLock : OFF we allow upper and lowercase
Start Col : 01 start column
Bell : OFF no right margin cell
XmitFnctn : NO move- and editor-keys are not transmitted to the computer
SPOW : NO spaces will overwrite existing characters
InhEolWrp : NO no wraparound at the end of line
Line/Page : LINE transmit one line at a time
InhHndShk : NO enable XON / XOFF handshake
Inh DC2 : NO ignored
Esc Xfer : NO transmit esc-sequences to the printer
TermMode : HP use HP special sequences
```

2. When you want to connect the terminal to an HP 2334A CLUSTER (PAD) you must change the following specifications:

```
DATACOMM:

 RecvPace  : Xon/Xoff               use XON/XOFF handshake
 XmitPace  : Xon/Xoff               on send and receive transmission.

TERMINAL :

 InhHndShk : YES                    disables the use of DC1-
 Inh DC2   : YES                    and of DC1/DC2/DC1-
                                    handshake on blockmode.
```

With the reconfiguration of the terminal you switch off the handshake normally used. For example: you do not use an XON and XOFF protocol between the computer and the terminal to control the dataflow on a direct line. The terminal does not transmit XON or XOFF (DC1 and DC3) automatically, this only can be sent by pressing the '**CTRL Q**' or '**CTRL S**' key combination.

The CLUSTER must be configured to use the PROFILE 1 to work properly.

The HP 3000 must have a PAD-terminal configured in the system like this:

```
DRT#        = ldev of inp        UNIT      = 0          CHANNEL = 0
TYPE        = 16                 SUBTYPE   = 0          REC.WTH = 40
DRIVER NAME = IOPAD0             DEV.CLASS = PADTERM
```

If you want to use VPLUS on the terminal connected to an HP 3000 via a port on a PAD (HP 2334A), you need a terminal supporting blockmode. For example:

```
HP 2382A, HP 2392A-2394A, HP 700/92-700/94 or similar.
```

One of the problems will be the transmission of a block after pressing the **ENTER** key. Normally the HP 3000 will control the dataflow with the normal DC1 and DC3 handshake (XON / XOFF). When starting VPLUS, the Application reconfigures the terminal, so it will use a handshake named DC1/DC2/DC1. By using this protocol, the computerside tells the terminal, that the FORM is transmitted completely and that the user may start entering data to the FORM. This is initiated with the signal DC1.

When the input is terminated either with an **ENTER**, a **SELECT** or a function-key, the terminal tells the ready-state of the data to the computer with the DC2-signal. The computer now will initiate the transmission with the signal DC1.

```
VPLUS                      TERMINAL                USER
-----------------------I------------I----------------------------
                       .            .
send a FORM ---------->.            .
                       .            .
FORM is ready,         .            .
data can be            .            .
entered                .            .
DC1------------------->.            .
                       .            .
                       .            .          The User can
                       .            .<--------- start typing.
                       .            .
                       .            .<------ ENTER or Function-key
                       .            .
                       .Data is ready
                       .to be       .
                       .transmitted .
<----------------------.DC2         .
                       .            .
VPLUS is ready         .            .
to receive the         .            .
data                   .            .
DC1------------------->.            .
                       .            .
                       .Terminal sends
                       .the data    .
                       .entered to  .
<----------------------.the computer.
                       .separated with 'us'
                       .and terminated with 'rs'
                       .            .
after receiving        .            .
the last block         .            .
VPLUS starts the       .            .
execution and          .            .
continues with         .            .
the next FORM--------->.            .
                       .            .
---------------------------------------------------------------
```

This handshake does not work when you use your terminal on a PAD. The PAD itself catches the DC1 and DC3 signals or generates them as needed. So you can not use them to specify the transmission between the computer and the terminal, or to use the DC1/DC2/DC1 handshake for blockmode applications.

The alternative is the use of the term=24 in the LOGON like this:

```
HELLO paduser,user/usrpass.account/acctpass;TERM=24
```

This terminal type does not exist as a Term-Type file. The only program that requires it as a parameter, is VPLUS (and all VPLUS using applications). VPLUS checks this parameter and if the terminal type is decoded to be 'type=24', then there will be a special handling of the FORMS.

Normally when starting the blockmode, the application will check the configuration of the terminal with the sequence 'esc^^' and 'esc~'. Additionally it will check if the terminal is able to SPOW. Thereafter it will switch the terminal with the escape sequence 'ecs&s0h1G'

to do the DC1/DC2/DC1-handshake.

When using TERM=24, the application first will set the InhHndShk=Y and the InhDC2=YES with the escape-sequence 'esc&s1h1G'. You should set this before you start the session, but if you forgot, VPLUS will do it for you (but other things could go wrong, so it's better to do it yourself to be sure!). This sequence will switch off all handshakes. After this, the terminal will be checked like usual. Additionally the terminal will be set to lock the keyboard every time the **ENTER** or one of the Function-Keys is hit. This is done with the sequence 'esc&k1K'. It will prevent the user from entering data before the new FORM is rebuilt completely, and the keyboard is unlocked with 'escb'. This is necessary because the normal handshake with DC1/DC2/DC1 does not work here.

```
 VPLUS Terminal User
------------------------I---------------I-----------------------
. .
the keyboard will be    .               .
locked, before the      .               .
transmission of a FORM  .               .
'esc c' --------------->.               .
                        .               .
now the transmission    .               .
of the FORM will start  .               .
FORM ------------------>.               .
                        .               .
when the FORM is ready  .               .
the keyboard will be    .               .
unlocked with           .               .
'esc b' --------------->.               .
                        .               . now the user can
                        .               . start to enter . .<---- data
                        .               .
                        .               .<- ENTER or function-key
                        .               .   this automatically
                        .               .   locks the keyboard . .
                        .start transmitting
                        .of the data    .
<---------------------- .entered        .
                        .separated with 'us'
                        .and terminated with 'rs'
                        .               .
when the FORM is        .               .
received completely,    .               .
VPLUS goes on and       .               .
continues with the      .               .
next Form ------------->.               .
                        .               .
                        .               .
and unlocks the keyboard.               .
after the the FORM      .               .
transmitted completely  .               .
'esc c' --------------->.               .
                        .               .
----------------------------------------------------------------
```

The difference between the use of VPLUS on a direct line and a PAD is just the kind of dataflow control. The problem comes up only by the intelligence of the CLUSTER CONTROLLER which tries to prevent his memory or that of the terminal from overflow.

This problem is solved by using ‘term=24’ in the logon. If you want to use VPLUS applications on a specified PADTERM, you can configure the terminal type in the system configuration to be ‘24’. This will switch on the correct blockmode handling for every user on this logical device.

Optimizing VPLUS Utilization

VPLUS, Hewlett-Packard's standard terminal management software, finds wide use in applications ranging from manufacturing control to financial transaction processing. Offered as part of the HP3000 Fundamental Operating Software (FOS), VPLUS provides programmers and designers access to the features of HP block-mode terminals, effectively insulating programs from the details of data communication and terminal control.

The Response Centers receive many questions regarding VPLUS optimization and program performance. This Note will address several topics related to these issues. There are three major areas where even small amounts of effort from the programmer can make significant differences in VPLUS performance:

- Effective forms design
- Stack use by VPLUS applications
- Programs' utilization of forms and forms files

A proper balancing of effort in all three areas can help you make significant strides towards efficient use of VPLUS.

A working knowledge of FORMSPEC and "general principles" of forms file management is assumed, as is knowledge of the major VPLUS routines (e.g., VGETNEXTFORM, VSHOWFORM, VOPENTERM).

Definition of Terms

In this Note we shall discuss what changes need to be made to the largest form in the forms file targeted as being troublesome. There are many valid definitions of "largest form". One is "that form in which the largest amount of data is transferred between the program and terminal". Another is "that form which contains the largest number of fields". A third is "the form which, when displayed on the terminal, uses up the most space on the "screen". When the term "largest" is used, it will be further defined for its context.

There are also three acronyms used throughout this paper. FST stands for Field Status Table, and is used by VPLUS to hold information about each field on the current form. DBUF and IBUF refer to the Data BUFfer and Input BUFfer used internally by VPLUS. Any program (through calls to VPUTBUFFER and VGETBUFFER) can manipulate the DBUF, while the IBUF is a reordering of the DBUF and is used internally by the VPLUS intrinsics for data transfer to and from the terminal.

Effective Forms Design

One of the easiest and least painful ways to help the performance of programs using VPLUS is to use foresight in designing the forms to be used by those programs. A few simple tricks applied in advance can save a lot of debugging and redesign later in the process.

One Screen Approach

One of the first things to keep in mind when designing forms is how much data the user needs to access at one time. Assuming there is enough information to fill the screen, does one want to show it all at once. Too much data may overwhelm the user, making data entry more difficult due to a crowded and confusing screen. Presenting data in small doses may frustrate the user as he moves through multiple forms to see all the data, making data entry that much longer.

Here, the old “space/time” tradeoff comes into play: by saving the space needed for the display of data (in the multiple-form scheme), more time is needed for showing all screens, while by displaying all the data at once, more space is needed (but all necessary information is shown in one pass). One advantage to the multiple-form method is that a user could stop viewing the screens of data once the needed data is displayed (assuming such an escape mechanism is designed into the program).

If you decide to use the one-screen approach, there are many things you can do to avoid problems before they occur. One is to keep the amount of data (in number of bytes) equal between forms in the forms file. This will enable VPLUS to get the greatest amount of utilization from the DBUF and IBUF, since there would then be only a few instances where just portions of each were needed for I/O operations. Another is to keep the number of fields as equal as possible between forms. This will allow VPLUS to get the greatest amount of utilization from the FST and other internal tables connected with field manipulation on each form.

Reducing the Size of Large Fields

If some forms in the forms file are composed of many small fields, while there is also a form with one or two VERY large fields (for example, a program where general information is needed about a person, but specific comments may be made), you might consider changing the form with those large fields into a form with smaller fields.

A good example of this is one where the entire screen is an unprotected field for text entry. That form would only need one entry from the FST (since there's only one field), but would require 80*24=1920 characters for the DBUF and IBUF. Changing this form to one which repeats, appends to itself, and is made of only one 80-character long field will allow the same functionality of the former form (with the exception of having to press **ENTER** at the end of each line), and actually be more flexible. This flexibility comes from the fact that now the user can input as many lines of text as are needed, whereas before they were restricted to 24 lines.

Similar things can be done to the form which has the largest number of fields. If the fields are some sort of tabular data, making another repeat/append type form set can help. In this set, the first form could contain header information, while the second is a self-repeating and self-appending form containing the fields for each detail line.

Using Forms Families

Another trick to speed up the display of forms is the effective use of forms families. Forms families are sets of forms which share a common screen definition, but different field characteristics. Any time a forms family member is to be displayed, VPLUS checks to see if the last form displayed was another member of that same family. If it was then instead of transmitting the entire form definition, VPLUS sends only those escape sequences needed to alter those fields that differ between the forms (e.g., from unprotected to display- only, from half-intensity inverse to underlined blinking, etc.). If all of the information to be displayed is similar in format, it's possible to use forms families to make minor changes in the display without repainting the entire screen. You can also use the intrinsic VCHANGEFIELD (released with version B.04.17 of VPLUS) to accomplish the same ends.

Combining Several Forms

If your program accesses several forms files, you should seriously consider combining them into a single file. Not only does this save stack space (since VPLUS must maintain separate control information-- including the DBUF and IBUF--on the program's data stack for each open forms file), but it helps make maintenance of the forms much simpler. If the forms are scattered among multiple forms files, the forms files must be closed and reopened each time they are used, and much of the time used by the program will be tied up in this moving between forms files, rather than with the actual processing of the data.

Stack Use by VPLUS Applications

Some of what will be talked about in this section will contradict what was discussed in the prior section. Once again, it is the computer world's old "space/time" problem: optimizing the data space required for a program will most likely increase the execution time of the resulting application.

If a VPLUS application aborts with any of the typical indicators of a stack problem (STACK OVERFLOW; a call to VOPENFORMF fails with error codes 40, 41, 61, 62, 68 or 69), the first thing to do is to compile the forms file into a fast forms file. One part of the space reserved on a data stack is an area which contains the directory of all records in a forms file. By all records, we mean ALL records-- source records containing the raw, input form definitions, code records containing the escape sequences needed to paint those forms on the screen, as well as FORMSPEC-internal records needed to manipulate those forms.

What goes into a fast forms file is only a few control records (to designate it as a fast forms file) and code records--but no source records. Since there are fewer records in the file, the directory is smaller and less space is needed on the stack for it. As a result, you save a minimum of 800 words per forms file simply by using fast forms files. (We should also note that, in addition to the space savings, fast forms files are so named because they require much less disc I/O. Some benchmarks have shown as much as a 50% reduction in forms file I/Os when a fast forms file is used instead of a slow forms file.)

Other methods of space optimization may involve taking out some of the features incorporated in the original design of the application. Local Forms Storage (LFS) is very useful for minimizing data communications overhead involved in VPLUS, since by its use a forms definition may be sent to a terminal once, but there is a price to be paid on the stack. If LFS is being used, a directory of form names already downloaded to the terminal is kept on the stack to enable VPLUS to quickly discover whether a form has already been downloaded. Each entry in this table is 16 words long. Therefore, if LFS is enabled for a large number of local forms (via VOPENFORMF), disabling it will save 16*n words, where n is that number of "local" forms. (Local forms storage will be discussed in greater detail in a later section.)

One of the ergonomic features discussed in the forms design section comes into play now. Remember, VPLUS keeps two copies of the data buffers on your stack (the DBUF and IBUF), in addition to any buffer space allocated by your program. Minimizing, or standardizing, the amount of data that is transferred for each form will help control the sizes of those buffers. For example, suppose most forms in the forms file contain 20 2-character long fields, while one form contains 40 6-character long fields. Whereas most of the forms would only require the DBUF and IBUF to be 40 characters long (20 fields times 2 characters per field), VPLUS will allocate 240 characters for both buffers (40 times 6). Changing that 40-field form to a self-repeat/append containing 4 fields brings that form's requirements down to buffers 24 characters long. Notice that by making that change, VPLUS's stack requirements have just decreased by 432 words (216 words each for the DBUF and IBUF).

In the same vein, if most of the forms have close to the same number of fields and almost the same data buffer requirements, but one form has MANY more fields, a simple splitting of the form into either a self-repeating of multiple forms (if dealing with REPEAT/APPEND is not desirable) will decrease the space needed for the FST, as well as the other internal tables needed to manipulate fields.

If stack space is a problem, and function key labels (FKLs) are enabled, their elimination could save up to approximately 800 words. When enabled, space is reserved for two copies of FKLs: one copy for global FKLs to be used in the forms file, and another to be used for FKLs local to each form. For each copy, there are 16 bytes of storage reserved per key label for the messages associated with each, as well as information used by VPLUS to tell if FKL definitions have changed (via calls to the VSETKEYLABELS or VSETKEYLABEL routines) and if those changed values have yet been displayed.

Another feature of VPLUS that occupies a large amount of stack space is the use of INIT, FIELD, and FINISH phase edit specifications. FORMSPEC compiles these into object code meaningful only to the appropriate VPLUS routines (VINITFORM, VFIELDEDITS, VFINISHFORM) which act upon the copy of the data brought into the IBUF by VREADFIELDS. Since each form can have up to 12000 words of code (which is treated on the stack as part of the form definition), either eliminating or simplifying edit specifications can save immense amounts of stack space.

All of these suggestions used in combination can save a program on the order of hundreds, if not thousands, of words of stack space. In most cases, simply using a fast forms file in combination with eliminating LFS will return more than enough stack space to allow processing to commence.

VPRINTSCREEN Intrinsic

A new intrinsic, VPRINTSCREEN, provides the capability to print the entire contents of a terminal screen during VPLUS execution. Currently, VPLUS provides the intrinsic VPRINTFORM which prints the current form with the data contained in the form data buffer; function keys, appended forms, line drawing characters, etc., are not printed. With VPRINTSCREEN, these limitations no longer exist.

The largest demand for this functionality has been from documentation and training departments. In the past, the only method for obtaining a complete "snapshot" of a screen was through the use of the internal printers available on some HP terminals.

There are two approaches a user can take to produce copies of VPLUS screens. One is to incorporate the use of VPRINTSCREEN into an end-user application so that screen images can be captured at run time. This allows actual screen and data to be captured in production mode. The other method is to develop a simple utility which would only be used to produce hard copies of screens. The utility should allow data to be entered into the screens before calling VPRINTSCREEN to produce the image. The advantage of this method is that it removes the overhead of using VPRINTSCREEN from the application, but still provides a mechanism for including reproductions of the screens and data in product literature.

The demo program, VPRTDEMO, is an example of this type of utility. Refer to the section "Viewing the VPRINTSCREEN Demo" for specifics on how to access and use the demo program. Any user interested in the VPRINTSCREEN intrinsic should use this demo with existing forms before attempting to modify existing applications.

The remainder of this article will concentrate on the process of calling VPRINTSCREEN from a user program. This procedure was implemented using the Pascal heap procedures to perform stack allocation. This introduces a high risk of conflict for applications which use the DLSIZE intrinsic for stack allocation. In the case of VPLUS, DLSIZE is used by the existing intrinsics when the application is written in COBOL, FORTRAN/66, or SPL. However, since VPRINTSCREEN always uses the heap procedures, applications written in one of these specified languages must follow two rules when calling VPRINTSCREEN.

1. The language id in the COMAREA must be set to 5
2. The INTRINSIC calling mechanism must be used when calling the VPLUS intrinsics from the main and all interacting parts of the application. For example, with COBOL the application must use:

   ```
   CALL INTRINSIC <intrinsic name>
   ```

Refer to Appendix E of the VPLUS Reference Manual, the *COBOL Reference Manual* and the *Pascal Reference Manual* for more information on these calling mechanisms.

For applications which use a language-id of 5 in the VPLUS COMAREA (Pascal, FORTRAN/77, HPBUSINESS BASIC), VPLUS uses the HEAP procedures for stack allocation. The new intrinsic can be called in the standard format from an application of this type.

VPRINTSCREEN

VPRINTSCREEN records the contents of the current screen to an off-line list device. A documentation option allows formatting for printing to a laser printer (See Modes of Operation). A VPLUS supported terminal is required during execution. TDP is required for the laser printer output.

```
    VPRINTSCREEN {COMAREA, READSIZE}
```

Parameters

COMAREA — The following COMAREA items must be set before calling VPRINTSCREEN (unless previously set):

CSTATUS	Set to zero.
COMAREALEN	Set to the total number of 2-byte words in COMAREA.
PRINTFILENUM	Set to the file number of the list file to which form is printed. If set to 0, VPRINTSCREEN opens the device "LP" as the list file, and sets PRINTFILENUM to the file number of the opened list file.

VPRINTSCREEN may set the following COMAREA values:

PRINTFILENUM	If VPRINTSCREEN opened the list file, set to the file number of the opened file.
CSTATUS	Set to a nonzero value if call is unsuccessful.
FILEERRNUM	Set to the MPE error code if an MPE file error is encountered.

READSIZE — 2-byte INTEGER. Reserved for system use; must be 0.

A possible implementation of this feature would be to define a function key which is available to the user for printing the screen contents in any transaction. This feature would be useful for providing immediate output of documents and data during production. This new intrinsic is demonstrated with the program VPRTDEMO. See the section titled "Viewing the VPRINTSCREEN Demo" in this note for directions on obtaining the demo.

Modes of Operation

VPRINTSCREEN operates in two modes; normal and documentation. Both of these modes are discussed in detail below.

Normal Mode

This is the default calling mode of VPRINTSCREEN. When called, the value is the PRINTFILNUM word of the COMAREA is used to determine the list device. If the calling program opens the list file, it must supply the file number of this file in PRINTFILNUM. VPRINTSCREEN opens the list file with the formal and actual file designator FORMLIST, assigns it to the device class LP, and specifies its length as 80 characters. This is equivalent to using the file equation:

```
   :FILE FORMLIST;DEV=LP;REC=-80
```

A user may change any of these characteristics with a `:FILE` command.

It is recommended that `VPRINTSCREEN` and `VPRINTFORM` not be used together within the same program. Since the same list file is used for both listings, output from the two calls will be intermixed.

Another visible difference between `VPRINTFORM` and `VPRINTSCREEN` is that `VPRINTFORM` gives you the option to underline fields. This option is NOT available with `VPRINTSCREEN` (LP mode).

A `PAGE EJECT` is performed each time `VPRINTSCREEN` is called (at the completion of the print operation).

Documentation Mode

The output provided in this mode of operation is intended for manual writers and/or programmers who are familiar with TDP. The user must already have TDP on their system or must purchase it to use `VPRINTSCREEN` in this mode.

In documentation mode, `VPRINTSCREEN` in conjunction with TDP, provides the capability to print screen contents on a laser printer (HP2680A and HP2688A). With this more, borders, field highlighting (other than color), alternate character sets, and active function keys are captured and converted to the appropriate font for printing on the laser printer. Documentation node is enabled by setting a JCW before running the program:

```
    :SETJCW VPRINTJCW=1
```

When `VPRINTJCW` is set to "1", the list file, `FORMLIST` is NOT opened. Instead, a temporary ASCII disc file, `EPOCLIST`, is created (or appended to, if it already exists). This file can be saved and renamed on completion, and then input to TDP and "finaled". A documenter can also add text to the file, or have a separate file, and use the TDP "include" statement to access the screens. (Refer to the *TDP Reference Manual* for a detailed discussion on USE and INCLUDE files).

In order to print the forms to a laser printer, an environment file must be created. (See the *IFS/3000 Reference Guide*). We have supplied sample environment files which can be used if the user does not already have his/her own files. If other environment files are used, they must include the font ids shown under `LIMITATIONS>` The supplied files are:

`VENV80`	environment file for the HP2680 laser printer
`VENV88`	environment file for the HP2688 laser printer
`VSETUP`	TDP include file which sets parameters for TDP listings and references the environment files
`VEPOCUSE`	TDP USE file which separates each screen contained in `EPOCLIST` into separate files
`VPRTDEMO`	Demo program which demonstrates the use of the `VPRINTSCREEN` intrinsic

NOTE See the section titled "Viewing the VPRINTSCREEN Demo" for information on obtaining these files.

Printing Screens from TDP

By default, `EPOCLIST` uses the `VSETUP` file to reference the supplied environment files. These files must be accessible by TDP before `EPOCLIST` can be printed. If a different environment file is used, `EPOCLIST` must be modified to reference the appropriate environment file. In addition, the font definitions from `VENV80` and `VENV88` must be included in the environment file used. (Refer to the fonts listed in the Limitations section).

Following are the steps involved to print the entire contents of EPOCLIST to a laser printer:

1. `:PURGE (or :RENAME)EPOCLIST`
2. `:SETJCW VPRINTJCW = 1`
3. :Run your application program.
4. Access the `PRINT` function wherever it's available to save screen contents in `EPOCLIST`. Remember, all the data entered by the user will also be saved as part of the screen.
5. Exit application.
6. `:SAVE EPOCLIST` (`EPOCLIST` is the temporary file which was created in the above step. It contains the contents of all the screens printed).
7. `:RENAME EPOCLIST, <new name>` If not renamed, the next time the application is run, the output will be APPENDED to the existing file.
8. `:RUN TDP.PUB.SYS`
9. Execute the following command:

   “`FINAL from <new name> to *HP2680`”

   or “`FINAL from <new name> to *HP2688`”.

   These final two steps could be specified in a job stream. The screens will be printed out one per page.

Merging Screens with a TDP File

Instead of printing the screens out separately, you may want to include them into an existing TDP file. A special file, `VEPOCUSE`, is provided to facilitate the user in this task. The `VSETUP` file must be included as one of the first statements of your TDP document file. It will set up the reference to the correct environment file necessary for printing the screens in `EPOCLIST`.

NOTE If you want to use a difference environment file, you must include the font definitions from `VENV80` or `VENV88` in your environment file. (Refer to the fonts listed in the Limitations section).

The first step is to separate the screens in EPOCLIST. VEPOCUSE can be used to do this by following these steps:

1. `:RUN TDP.`

   Clear workspace.

2. `USE VEPOCUSE.`

   Answer the prompts: filename, number of occurrences, and desired PREFIX. VEPOCUSE will then split the file into 'n' files, all prefixed with the PREFIX entered.

3. Text in your document, and include the appropriate screen file in the appropriate location.

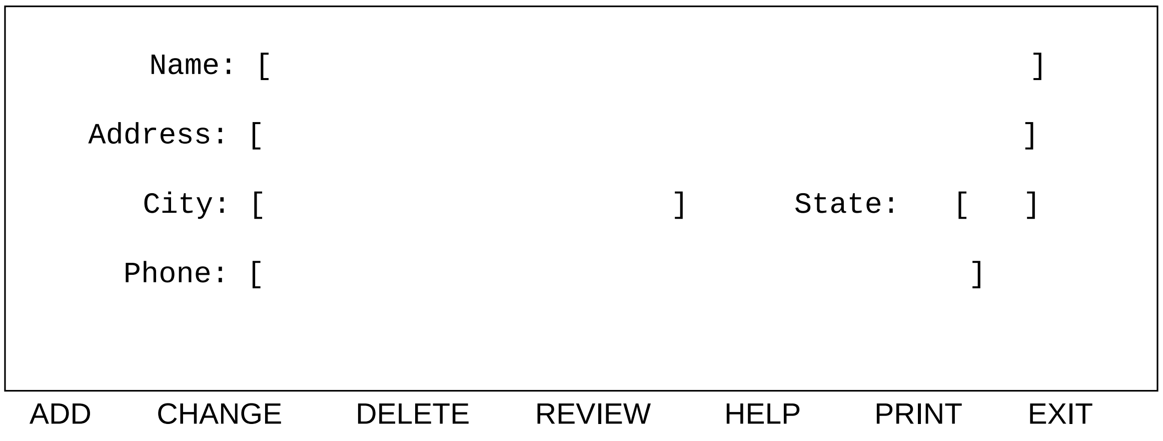
It is recommended that the screen files are "included" into the file, because EPOCLIST has a record size of 168 bytes, and most document files are set to 80 bytes.

Sample EPOCLIST File

```
*>>DATE:FRI,MAY 3, 1986, 11:19AM
if main in hpvsetup.pub
image 28
need 28
>>>>>screen formatting commands<<<<<
*>>END VPRINTSCREEN B.04.20
```

Sample EPOCLIST

Figure L-4. Sample EPOCLIST Screen from TDP



```
      Name: [                                        ]

   Address: [                                        ]

      City: [                     ]      State:  [   ]

     Phone: [                                     ]
```

ADD   CHANGE   DELETE   REVIEW   HELP   PRINT   EXIT

Limitations

The Intrinsic VPRINTSCREEN

- Procedure calls must be modified in order to call the intrinsic from a language which does not use a VPLUS language id of 5 in the COMAREA. (COBOL, SPL, FORTRAN/66).
- Uses additional stack resources.

The Environment Files

- Native Language support is NOT available for VPRINTSCREEN. To print a screen in another language, a LOCALIZED environment file must map to the following fonts:

  c = full bright

  g = half bright inverse video

  d = normal l = line draw

  m = math

- There is no distinction between fullbright and halfbright provided with the current environment files. However, the code is set up to distinguish between them. (See the above fonts). Currently, the fontids "c" and "g" map to the same font in the VSETUP file.
- Screen images cannot be scaled. In other words, fonts are provided in one size only. A "scaled font" environment file which maps to the same fonts above, would need to be created.
- Sample files are provided. They can be modified or the user can create their own using the font ids defined above.

Use of TDP

- The maximum TDP record size is 168 characters. A single line of a display screen can contain multiple escape sequences, and easily exceed this limit (e.g., line draw, full bright, many single character fields). If this limit is exceeded, the line may be truncated.
- TDP Macros '5' through '9' are used to alleviate this truncation, but some screens are complicated enough to reach this limit anyway. If you use your own macros 5-9 (other than as temporary macros), some inconsistencies may exist when merging text and screens.
- TDP may indicate errors have occurred, when in fact, there are no errors. The most common messages are: "Unrecognizable command" and "x Characters have been truncated".

Viewing the VPRINTSCREEN Demo

The new VPLUS intrinsic, VPRINTSCREEN, can be demonstrated with the program VPRTDEMO. You can simply run VPRTDEMO and specify a forms file name. The application will present the forms in alphabetical order and call the VPRINTSCREEN intrinsic when the user selects the **PRINT** function key, F3. At the beginning of execution, the user will have the option to automatically print all forms in the forms file.

Several sample and demo files have been supplied as examples for the user. They are contained on the installation tape in the HPPL89 account and HP32209 group. They should be restored to a local group and account. Some name modifications may be necessary (see below).

VENV80	This is the environment file required for printing to the HP2680 printer.
VENV88	This is the environment file required for printing to the HP2688 printer.

`VSETUP`	This is a TDP "include" file which selects and sets up the proper environment for the printer selected. This file should be modified so the references to `VENV80` and `VENV88` are properly qualified by the `GROUP.ACCOUNT` they reside in. If the document you are printing resides in a separate group and account from `VSETUP`, then a file equation must be issued to reference `VSETUP`. This also applies if the user renames the `VSETUP` file to a local name. In either case a file equation must be issued, for example: `:FILE VSETUP = VSETUP.MYGROUP.MYACCT -OR- :FILE VSETUP = MYSETUP.MYGROUP.MYACCT`
`VEPOCUSE`	This is a TDP "use" file, potentially needed when merging text with screens.
`VPRTDEMO`	This is a demo program which demonstrates the use of the `VPRINTSCREEN` intrinsic. It can be run in normal or documentation mode. Following are instructions for running the demo.

Running VPRTDEMO

1. `:PURGE (or :RENAME) EPOCLIST`
2. `:SETJCW VPRINTJCW = 1`
3. `:RUN VPRTDEMO.`
4. Accessing the `PRINT` function wherever it's available to save screen contents in `EPOCLIST`. Remember, all the data entered by the user will also be save as part of the screen.
5. `Exit VPRTDEMO`.
6. `:SAVE EPOCLIST` (`EPOCLIST` is the temporary file which was created in the above step. It contains the contents of all the screens printed).
7. `:RENAME EPOCLIST`, <new name> If not renamed, the next time the application is run, the output will be APPENDED to the existing file.
8. `:RUN TDP.PUB.SYS`
9. Execute the following command:

   "`FINAL from <new name>to *HP2680`"

   or "`FINAL from <new name>to *HP2688`".

   These final two steps could be specified in a job stream. The screens will be printed out one per page.

If you answered "Y" to the automatic print prompt, the forms will be displayed and printed in alphabetic order. At completion, you will have `EPOCLIST`. If the `VPRINTJCW` is not set, then a `FORMLIST` file will be created.

If manual print mode was selected, then the first form in alphabetic order is displayed. If local function keys were defined for the form, then these keys will be displayed.

However, the action performed by `VPRTDEMO` when a key is selected is defined below:

Key 1 = FIRST FORM	<displays first form in file>
Key 2 = not defined	
Key 3 = PRINT	<prints screen design to `EPOCLIST` or `FORMLIST`>
Key 4 = REFRESH	<refresh current screen design>
Key 5 = PREV FORM	<displays previous form in file>
Key 6 = NEXT FORM	<displays next form in file>
Key 7 = SELECT FORM	<prompts for name of form to be displayed>
Key 8 = EXIT	<exit `VPRTDEMO`>

M Application Notes

This appendix contains a number of application notes which are also posted on the KMINE web site. They represent some of the most frequently asked questions (and the answers) about VPLUS . There are hundreds of similar notes and articles posted to KMINE.

KMINE is a large database of problems and solutions which is accessible by both HP staff and customers. It is searchable by key word topics. You can also define your search more narrowly by specifying the types of documents you wish to search.

If you have not used KMINE in the past, you will need to register your ID and password. The KMINE web site is located at the following URL.

http:\\12485kmi.mayfield.hp.com/kmine

Workarounds for VPLUS Forms Fille 32767 Record Limit

Problem Description

When trying to add a form to my forms file in FORMSPEC, I get an INTEGER OVERFLOW error. I understand that this is due to the forms file record limit of 32767 (see RCEN [W1331556/RCEN/English]). How can I add my new form ?

Solution

There are three solutions to the 32767 record limitation for VPLUS forms files:

1. Use two forms files. The disadvantage to this solution is the need to change any existing applications to use the two forms files. In addition, there will be some overhead when closing one forms file and opening another.
2. f you have ever deleted forms from the forms file, you should be able to gain some space by creating a new forms file and copying the forms from the old to the new forms file. This can be done online or via VPlus batch commands as documented in Chapter 7 of the HP Data Entry and Forms Management System (VPLUS/V) Reference Manual
3. .Use two forms files as source and have a single target fast forms file using the new VMERGE utility available on the the latest VPLUS patch on both MPE/V and MPE/iX. This utility takes advantage of the fact that you can fit a lot more forms into a fast forms file than into a regular forms file.
4. This solution would require the ongoing use of two regular forms files for development, but the applications could continue to use only the single merged fast forms file. The VMERGE utility is documented in the MPE/iX 5.0 and MPE/V 3P Communicators.

HP Only Info

Additional note on solution #2 to copying forms to a new forms file:

2A. VCOPY job stream is an UNSUPPORTED job that can be given to the customer to automate this task. This job stream copies all active forms and relates parent/child forms. It can be found in Mountain View on Spam as VCOPY.VPLUS.BLOECHL. Instructions are included.

2B. See instructions provided in SR 5003126110.

Using the VPLUS Environment Control File (VENVCNTL)

Problem Description

What is the VENVCNTL.PUB.SYS file on my system, and what is it used for?

Solution

The VPLUS environment control file, VENVCNTL, allows applications to override specific VPLUS defaults. VPLUS always checks for the existence of VENVCNTL.PUB.SYS or, alternatively, the existence of a file equated to this formal file designator. If no VENVCNTL file exists, no VPLUS defaults are overridden.

VENVCNTL is a simple text file containing only one line of data. Each override option may be activated by setting the option byte number to 1. For example, in the VENVCNTL file below, options 1, 5, and 20 are set.

```
:print venvcntl
10001000000000000001
```

For MPE V/E, the basic recommendation is either no VENVCNTL file, or, if there are no 264x terminals capable of accessing the system, a VENVCNTL file with byte 1 set.

For MPE/iX, the basic recommendation is a VENVCNTL file with bytes 1, 5, and 20 set. Since 264x terminals will not work on MPE/iX, byte 1 should always be set to improve performance.

The table below explains the function and possible side effects of each VENVCNTL option. Note that many bytes are either diagnostic only or reserved for future use; these bytes should never be set to 1.

Byte	Explanation	Side Effects / Comments
1	VOPENTERM does not try to determine terminal identity if terminal does not have self-identification capability.	VPLUS will not run on terminalsthat cannot self-identify (264x terminals).
2	VPLUS does not lock out terminal Aids/Modes/User keys during interactions.	User could corrupt terminal configurations established and expected by VPLUS.
3	Reserved.	
4	VPLUS bypasses Modified Data Tag (MDT) feature of MDT terminals.	MDT terminals will transmit all input field data, not the subset that user actually keyedin; increases datacomm and associated processing overhead.
5	(MPE/iX only) VPLUS strips carriage return character from terminal status read input.	DO NOT USE this option on MPE V/E; MPE V/E driver already strips the carriage return characters so this option is redundant on MPE V/E.

Byte	Explanation	Side Effects / Comments
6	(MPE V/E only) VPLUS extends terminal status reads to ensure that input termination character is accounted for.	DO NOT ACTIVATE this option unless experiencing VPLUS ”terminal status request failed” errors due to network delays.
7	Diagnostic only.	
8	VPLUS does not force 7009x terminals to 80 columns.	If 7009x terminal is configured for 132 columns, there is a possibility of data corruption.
9	VTURNON and VTURNOFF do notreset terminal function keys to default strings.	
10	(MPE/iX) VPLUS inhibits dual terminator; only the record separator (instead of either the record separator or the carriage return) will be recognized as a record terminator.	It will be awkward for user to recover a blockmode application after a power failure. Also, applications will treat line feeds as terminators, not as data.
11	VTURNON does not save 7009x terminal function key configuration information.	
12	VOPENTERM does not save and VCLOSETERM does not restore 7009x terminal function configuration information.	
13 - 18	Reserved	
19	Diagnostic only.	
20	(MPE/iX only) VPLUS posts pending terminal driver reconfigurations to overcome timing windows and force synchronization.	
21 - 29	Reserved	
30	VPLUS posts pending terminaldriver reconfigurations to overcome timing windows and force synchronization in a multi-hop network environment.	There will be incremental file system overhead associated with the posting action; this overhead is only necessary if difficulties are being experienced.
31	VOPENTERM and VIDTERM do not check JOB/SESSION mode when doing terminal open (available in soon to be released VPLDV28 (MPE V/E) and VPLEXF7 (MPE/iX) patches).	If executing VPLUS in a job and no valid file equation to the terminal exists, then job $STDIN file may be corrupted.
32 - 80	Reserved	

How to Trace VPLUS calls in the Program While it is Running

Problem Description

How can you track the VPLUS calls in the program as they are being

executed?

Solution

To produce a record of VPLUS calls from a program as they are called, use the following:

```
:SETJCW VIEWTRACE=1
:FILE TRCFILE;DEV=LP
:RUN progname;STDLIST=*TRCFILE
```

The file equation for TRCFILE could as easily be used for a disc file or for display to another terminal by doing the following

```
:FILE TRCFILE;DEV=DISC  (for disc)
```

or

```
:FILE TRCFILE;DEV=nnn   (nnn being the ldev number of the terminal)
```

How to Redirect $STDLIST in a VPLUS Application

Problem Description

Whenever my VPLUS program gets error messages those messages are displayed in the unprotected fields on the screen. Why does this happen? How can I redirect the messages so that they are either displayed on the printer or written to another file?

Solution

These messages are being displayed on the screen because the error messages are written to the STDLIST, which by default is the terminal. When a VPLUS program is run, the error message will be displayed in the unprotected areas of the terminal.

To redirect the STDLIST so that the messages will be displayed to the printer, use these steps:

1. `:FILE LP;DEV=LP`
2. `:RUN PROG;STDLIST=*LP`

To redirect the STDLIST so the the messages will be written to file, use these steps:

1. :BUILD OUTFILE;REC=-132,,F,ASCII;DISC=10000
2. :RUN PROG;STDLIST=OUTFILE

FORMSPEC Gives FS Error -99 at Compile Time

Problem Description

When trying to compile my formsfile, I receive the following error:

```
  FS ERROR -99
```

What does this mean?

Solution

The FS error -99, is a FormSpec error. This error means your formsfile is at end-of-file (eof). To verify this, do a :LISTF formsfile_name,2 and look at the EOF and LIMIT. If they are the same or very close, this is the reson you are getting this error.

The maximum size of a VPLUS formsfile is 32767. If your formsfile is less than this size, let's say around 28000, you can try the following: The first step is to build a new larger formsfile:

```
:BUILD MNTFORM1;REC=128,1,F,ASCII;DISC=32767;CODE=VFORM
```

Then do the following:

1. :FCOPY FROM=oldform;TO=newform
2. :RENAME oldform,something
3. :RENAME newform,oldform   (:RENAME newforms file to original's)

Index

Index

Index

Index

Index