VFORM

Table of Contents

Section 1

Introduction and Overview

Introduction

The virtual forms or vform_ package is designed to provide an asynchronous, device independent forms capability for a wide variety of terminal types. This is accomplished by providing a terminal controller for the various different terminal types supported. This terminal controller provides information necessary for the various terminal functions and screen attributes. Limited support, that is, support without any screen attributes such as inverse video, etc., is provided for any terminal capable of running with the Multics video system. A hardware forms capability is not required and is not utilized if it exists. It is a totally asynchronous, software driven package.

Use of the form software is divided into two major areas: form definition and form manipulation. Each of these areas is discussed below.

Form Definition

A form is defined by specifying the various properties of the form using "form" statements and "field" statements. The details of these statements are discussed in Sections 2 and 3 respectively. The "form" statements specify information that applies to the form as a whole such as the height and width of the form. The "field" statements specify information that is relevant to each field of the form such as its location on the screen, its screen attributes, whether or not it is protected, etc. Fields can even be defined "like" other fields.

A form definition segment must have a ".form" suffix, and is translated to a machine readable segment of the same name as the form definition segment but without the ".form" suffix (see Form Translation below). For example, for a form called "input", the form definition segment would be called "input.form", and the machine readable form segment would be called "input". The form designer should be careful to avoid conflicts when naming the form, the database, the programs using the forms, etc.

Form Translation

Once all of the information about a form has been defined in the form definition segment, it may be translated to its machine readable counterpart using the translator program "cv_form". The "cv_form" translator produces the machine readable form segment with the same name as the form definition segment but without the ".form" suffix.

This machine readable form segment defines the initial state of the form whenever the form is opened.

Form Manipulations

Once a form is defined and converted to a machine readable format, there are numerous operations that the applications programmer can perform on the form. A typical scenario is for an application program to:

1) open the form (vform_$open)
2) display the form (vform_$display_form)
3) read the form (vform_$read_form)
4) interpret the data (part of applications program)
5) display the form again if step 4 could affect the screen in ways other than using the form (vform_$display_form)
6) go to step 3 if not done

Normally, an application displays a form once, and does not use the vform_$display_form subroutine unless the screen image has been destroyed by output external to the forms software, such as error messages, questions, etc. This is because displaying the screen requires considerable time, both processing time and real time. Calling the vform_$read_form subroutine updates the screen with any changes that may have been made to the fields of the form since the last call to vform_$read_form, vform_$update_screen, or vform_$display_form. Such changes to the fields of a form could have been made with the vform_$set_value, vform_$set_attributes, etc. The vform_$update_screen subroutine will force a screen update of all changes to the screen, but its use is not usually needed since the next time vform_$read_form is called, it updates the screen. The vform_$update_screen subroutine should only be used when some information must be presented to the user before the next vform_$read_form operation.

Other form manipulations are performed during the vform_$read_form. At this time, control is taken from the applications program, and the forms package is in control. For each field defined in the form, the designer may specify a "check_proc". This is a procedure (program) that will be called when the user leaves the field for any reason. The program may validate the data in the field and may change the values or attributes of the current field or any other field of the form to convey an error message to the user. Entry points into the vform_ subroutine are provided for these manipulations. A "check_proc" may signal the "abort_form" or "exit_form" conditions to simulate the user pressing the appropriate key sequence or function key.

A provision similar to the "check_proc" is provided for use when exiting a form. This is called an "exit_proc" and may be used to ensure that various relationships between fields are maintained. For example, an "exit_proc" could enforce the rule that if field A is given a value, then field B must also be given a value or that the date in field C must be less than the date in field D. It may report errors in the same way as a "check_proc".

A description of the "exit_proc" and "check_proc" statements is provided in Sections 2 and 3 respectively.

Other manipulations are done at various times to change the attributes of fields, the values of fields, and to activate or deactivate fields on the screen. These, and all manipulations performed by the vform_ subroutines, are performed on a copy of the data. The converted, machine readable form segment always defines the initial state of the form whenever it is opened, and can only be changed by modifying the ".form" source segment and reconverting it. Section 4 describes the various vform_ subroutines used for these manipulations.

Section 2

Form Definition Statements

The first section of a form definition segment contains statements that describe the various properties of the form. These statements describe properties such as the name of the form, the location of the form on the screen, the size of the form, and a procedure to be called to validate the data in the form when the form is exited by the user. The form definition segment is created by the applications programmer using any text editor and is converted to a machine readable, binary form segment by a program called "cv_form" described in Section 5 of this manual.

When the various form and field statements are referred to, the following format is used:

"‹name-of-statement›" ‹type-of-statement› statement

Such as:

"row" form statement

Which refers to a statement that defines the "row" property of the form. This format allows the above statement to be distinguished from the statement:

"row" field statement

which refers to a statement that defines the "row" property of a field. This convention is adhered to throughout this manual.

A "form" form statement is required to be the first statement in a form definition segment and an "end" form statement must be the very last statement of the segment.

Each statement in the form segment begins with a keyword, followed by a colon (":"), followed by the information pertaining to the keyword (sometimes required to be in quotes), and ending with a semicolon. The exact syntax of the various form and field statements is given below. The "Notes" section for each statement indicates whether or not a given statement is optional and states the default value, if any.

column Form Statement:

Syntax:

```
     column:  ‹unsigned_decimal_integer›;
     col:  ‹unsigned_decimal_integer›;
```

Example:

```
     column:  1;
     col:  1;
```

Description:

The "column" form statement is used in conjunction with the "row" form statement to define the origin of the form on the screen. This field defines the column on the screen where the form is to begin. All fields defined within the form will have their column positions placed relative to the column defined in this statement.

Notes:

The "column" form statement is optional, and the default column is 1.

The "column" and "row" form statements are designed to allow a programmer to define multiple forms that can be active and on the screen simultaneously.

This feature is not currently implemented. Currently, the form column statement is checked for validity and ignored.

end Form Statement:

Syntax:

```
     end;
```

Description:

The "end" form statement specifies the end of the form definition segment. This statement is required to be the last statement in a form definition segment.

Notes:

Realize that this statement is the only "form" statement that is not placed at the beginning of the form definition segment. It must be the last statement in a form definition segment or an error will occur in the translation.

exit_proc Form Statement:

Syntax:

```
     exit_proc:  "‹Multics_pathname›";
```

Example:

```
     exit_proc:  "›udd›UserProj›UserName›form_utils_$input_form_exit_pro
```

Description:

The "exit_proc" form statement is used to define an application specific program that is to be called whenever the user transmits the form. This procedure may be used in conjunction with "check_proc" field statements for the various fields that are defined in the form to insure that all data is valid before the user is allowed to exit the form.

Notes:

The "exit_proc" form statement is optional. If an "exit_proc" form statement is not specified, then no application specified procedure will be called when the user transmits the form.

The "‹Multics_pathname›" must be a quoted string and may be either a relative or absolute pathname. If the pathname is a relative pathname (begins with a "‹"), the exit procedure is searched for in the directory specified relative to the current working directory. If the pathname is absolute (begins with a "›"), the exit procedure is searched for in the directory specified. If the pathname is simply an entry name (no "›" or "‹"'s), the exit procedure is located using the vform search paths. See the add_search_paths command in the manual AG92, Multics Commands and Active Functions, for more information about search paths.

It is recommended that a standard system error code be returned even though no action is currently taken to convey any information other than "beeping" the user's terminal. This is because it is planned that at some time in the future, a "status line" may be implemented as one of the screen lines that will display these messages in their text form. An error code called vf_et_$invalid_data whose message is "Invalid data for this field" has been provided for this purpose. For a discussion of creating customized error codes and a partial list of available error_table_ codes along with their text messages, see the manual AG91, Multics Programmer's Reference Manual, Section 7.

An "exit_proc" is called with three arguments: the form index of the form, the name of the form, and a standard system error code. A non-zero error code indicates that the data did not meet the requirements of the exit_proc, and the user is not allowed to exit the form until the data in error is corrected.

It is the responsibility of the "exit_proc" to inform the user of the nature and location of the error that causes the exit_proc to return a non-zero error code. An "exit_proc" may inform users of errors in the form by using the various vform_ subroutine entry points to manipulate the attributes of the fields in the form. For example, if field A is to contain a date which must be less than the date in field B, then "exit_proc" may activate an error field and set its value to something like "Date in field A must be less than the date in field B". The "exit_proc" may do other things like set the offending fields to inverse video, or change the value and/or attributes of any field of the form.

The vform_ software calls the "exit_proc" internally, so it should be written as to accept parameters as if it were declared and called as follows:

```
     dcl  exit_proc entry (fixed bin(35), char(*),
          fixed bin(35));

     call exit_proc ((form_index), (form_name), code);
```

This is not to imply that the applications program needs to call the exit procedure. This information is provided to describe the interface used by the vform_ software to call the exit_proc so that the application programmer will know how to design the exit procedure. The exit_proc should expect these parameters and handle them accordingly.

The form_index and form_name are passed by value (in parenthesis) so that any modifications made to these parameters to the "exit_proc" are ignored. Notice that the code parameter is not passed by value but by reference since the "exit_proc" is allowed to modify its value. See AM83, Multics PL/I Reference Manual, Section 12 for a discussion of passing parameters by reference and by value. See the vform_$open subroutine entry point described in Section 4 of this manual for more information about a "form_index".

__form__ Form Statement:

Syntax:

```
     form: ‹valid_form_name›;
```

Example:

```
     form:  input_form;
```

Description:

The "form" form statement defines the beginning of a form in the form definition segment. It is required to be the first statement in a form definition segment. If this statement is not the first statement in a form definition segment, then all statements in the form definition segment are ignored until a "form" statement is found.

A ‹valid_form_name› is a character string of 32 or less characters, but cannot be the null string. The first letter of the ‹valid_form_name› must be alphabetic, and all subsequent characters (if present) must be alphabetic, numeric, or the special character "_".

Notes:

The "form" form statement is required.

The "form" form statement is designed to give a form a name, and to separate multiple forms should they be defined within the same form definition segment.

The feature of allowing multiple forms within the same form segment is currently not implemented. Therefore, each form must be defined in a separate segment.

A good rule of thumb to follow when choosing the name of a form is the name of the segment minus the ".form" suffix. Remember that the vf_create_include_file uses the name specified by the "form" form statement when choosing its default output file name.

height Form Statement:

Syntax:

```
     height:  ‹unsigned_decimal_integer›;
```

Example:

```
     height:  23;
```

Description:

The "height" form statement describes how large the form is on its vertical axis, that is, how many lines on the screen the form may occupy. It is used to be sure that the form will fit on the screen of the intended terminal, so its value should not be larger than the number of lines on the smallest terminal the form may possibly be used with.

An error is reported if the row specified for any field of the form is greater than the value of the height of the form.

Notes:

The "height" form statement is optional and the default value is 20.

row Form Statement:

Syntax:

```
     row:      <unsigned_decimal_integer>;
```

Example:

```
     row: 1;
```

Description:

The "row" form statement is used in conjunction with the "column" form statement to define the origin of the form on the screen. The "row" form statement defines the row on the screen on which the form is to begin. All fields defined for the form will have their row positions defined relative to the value of this statement.

Notes:

The "row" form statement is optional and its default value is 1.

The purpose of the "row" and "column" form statements is to allow multiple forms to exist on the screen simultaneously. This feature is not currently implemented. Currently, the "row" and "column" form statements are checked for syntax, but ignored.

width Form Statement:

Syntax:

```
     width:  <unsigned_decimal_number>;
```

Example:

```
     width:  60;
```

Description:

The "width" form statement describes the number of columns used in the form. The value of this statement is used to insure that the form will fit on the screen of the intended terminal, and should be set to the number of columns of the smallest screen that the form may be used on.

An error is reported if part of any field extends beyond the column specified by the "width" form statement.

Notes:

The "width" form statement is optional and the default value is 79 columns.

Section 3

Field Definition Statements

The second section of a form definition segment contains a description of each field in the form and all of its properties. This section of the form definition segment begins at the first "field" statement. It ends at the end of the form definition segment with an "end" form statement. See the description of the "end" form statement in section 2 for details. Only one "end" form statement is allowed, and it must be the last statement of the form definition segment.

When the various form and field statements are referred to, the following format is used:

"‹name-of-statement›" ‹type-of-statement› statement

Such as:

"row" form statement

which refers to a statement that defines the "row" property of the form. This format allows the above statement to be distinguished from the statement:

"row" field statement

which refers to a statement that defines the "row" property of a field. This convention is adhered to throughout this manual.

Each field definition consists of a "field" field statement followed by various other field statements that describe the name, location, and other properties of the field.

The field statements allow the designer to define the current field "like" another field. When this is used, the field that the current field is "like" must be previously defined in the current form, and the definition must have been valid. If not, an error will occur when the form is translated to its binary form using the cv_form command. It is likely that one error can cause many error messages if fields are defined "like" other fields. See the description of the "cv_form" command in section 6 for more information.

If a field or one of its properties is defined "like" another field, the new field or property takes on all of the attributes of the field or property which it is "like". Defining a field "like" another field basically just sets the default properties of the new field to be the same as the properties of the old fields. These "defaults" may, of course, be overridden by simply redefining any property with the appropriate field statement. Defining a property "like" the corresponding property of another field copies the property from the previously defined field into the new field. The only exception to this is the "active" attribute (see the attribute field statement below). A field is always considered "active" unless explicitly defined otherwise using an "attribute" statement. Any changes to the properties of a field which is "like" another field replace the values set up by the "like", that is, specifying a "check_proc" for a field which is like another field will replace the "check_proc" obtained by the "like" attribute, not called in addition to the "check_proc" of the field that the current field is "like".

Certain field properties are "required" to insure that the information pertaining to a field is valid. Whether or not a field is required is described in the Notes section of each statement along with the default, if any. It should be noted, however, that if a field is defined "like" another field, then even these required properties are obtained from the "like" field.

Remember that an "end" form statement must be present after all fields have been defined. See page 2-3 for a description of the "end" form statement.

field Field Statement:

Syntax:

```
     field:  <field_name>;
     field:  <field_name> like <previously_defined_field>;
```

Example:

```
     field:  date;
     field:  date like first_date;
```

Description:

The "field" field statement is used to begin the definition of a field in a form. A "field" field statement must be the first statement after the various properties of the form are defined (See Section 2). A "field" field statement is also used to separate each field in the form, that is, a "field" field statement marks the beginning of the definition of a new field.

A field may be defined to be "like" another field in which case the new field takes on all of the properties of the field which it is "like" which are not explicitly redefined. The only exception to this is the "active" field attribute (See the "attribute" field statement below). All fields are active unless specifically defined otherwise in an "attribute" field statement.

A field name is a character string of 32 characters or less, the first of which must be an alphabetic character. Any additional characters, if present, must be alphabetic, numeric, or the special character "_".

Notes:

The "field" field statement is required for each field defined in the form.

Once the first "field" field statement has been encountered, none of the properties of the form (as described in Section 2) may be defined or changed.

attributes Field Statement:

Syntax:

```
     attributes:  ‹attribute_list›;
     attribute:  ‹attribute_list›;
     attr:  ‹attribute_list›;
     attributes:  like ‹previously_defined_field›;
     attr:  like ‹previously_defined_field›,‹attribute_list›;
```

Example:

```
     attributes:  underlined,^protected,active,inverse;
     attribute:  underlined,^protected,active,inverse;
     attr:  u,^prot,active,inv;
     attributes: like first_field;
     attr:  like field_x_title,active,prot;
```

Description:

The "attributes" statement describes the various attributes of the field. These attributes describe the characteristics of the field. Below is a list of the attributes that are available. An attribute may be negated by preceding it with a "^" (caret) character. The items of an attribute_list are separated by commas and must be terminated with a semicolon.

Field attributes may also be defined "like" those of another field, optionally adding other attributes as shown above.

The attributes that a field may have are listed below. Short names or abbreviations (if any) are listed in parentheses.

active (act, a): Whether or not the field will appear when the form is displayed. For example, fields used only to display error information would be inactive until the error occurs. This may be accomplished by using check_proc's for the field or exit_proc's for the form.

protected (protect, prot, p): Whether the data in the field is protected or whether it may be modified. When moving the cursor around on a form, it is not possible to position the cursor in a protected field. However, routines such as check_proc's or exit_proc's are free to modify the values of these fields.

hold_at_end (hold): If a field has the "hold_at_end" attribute, then the forms package will not automatically tab to the next field when the field is full, that is, after inserting data into the last character position of a field. The default mode is "^hold_at_end", which means that, by default, once data has been placed in the last character position of the field by the user, the forms package automatically tabs to the next active, ^protected field.

overflow (over): If a field has the "^overflow" attribute, then the user is not allowed to insert characters out of the end of the field. This is useful to insure that data is not lost in text fields. When an overflow of data does occur, the sub_error_ condition is signaled. The application program may trap this condition and handle the condition appropriately. See the "Notes" section for more information.

The following attributes are available to the form designer, and are available only when the terminal is able to support the attributes:

blinking (blink, b)
dotted_underlined (dotted_underline, du)
half_bright (half, h)
inverse_video (inverse, inv, i, reverse_video, reverse, rev, r)
underlined (underline, under, und, u)
vertical_separator (vs)

Notes:

The "attributes" field statement is optional, and the default is:

```
     attributes:  active,^prot,^b,^du,^h,^i,^u,^vs,^hold,overflow;
```

When the "^overflow" attribute is specified for a field, and the user attempts to make an insertion into a field that would cause data to be pushed out of the field, the sub_error_ condition is signaled so that the applications program can be notified of this event. The sub_error_ condition is signaled with the error code vf_et_$field_overflow and the condition name "field overflow". This condition is also signaled with the quiet_restart flag enabled which means that if the application does not intercept the condition, then it will simply be ignored. See AG91, Multics Programmer's Reference Manual, section 7 for a discussion of conditions and their handling.

charset Field Statement:

Syntax:

```
     charset:  <quoted_string_or_valid_keyword>;
     charset:  like <previously_defined_field>;
     charset:  like <previously_defined_field>,<quoted_string_or_valid_k
```

Example:

```
     charset:  "1234567890";              /* a numeric field */
     charset:  upper;                     /* UPPER CASE field */
     charset:  numeric," .e+-";           /* Numeric, blank, ".e+-" */
     charset:  like other_data;           /* Same as field "other_data */
     charset:  like other_data," ";       /* Same as "other_field" plus ".
```

Description:

The charset field statement defines the allowable set of characters for a field. The argument to the statement is a quoted string that contains the set of allowable characters or a designated keyword (described below). If a user types a character not in the defined character set while in the field, the terminal's bell is rung and the character is not entered into the field.

Notes:

The default is no "charset" which accepts all characters.

Certain keywords are provided for ease of use for often used character sets. The acceptable keywords are described below with their short names or abbreviations (if any) in parentheses.

alphabetic (alphabet, alpha, a): The field must contain only lower or upper case characters (a-z and A-Z).

alphanumeric (an): The field must contain upper case characters, lower case characters, or numbers. Note that this does not include a sign or decimal point, thus only unsigned integers are allowed in addition to the upper and lower case characters.

lower (l): The field must contain all lower case (a-z) characters.

numeric (numbers, n): The field may only contain numbers. Note that this does not include a sign or decimal point, thus only unsigned integers are allowed when this keyword is specified.

text (printable_chars, printables, any, ascii): This allows any printable, ASCII character. This is the default if no charset statement is given.

upper (u): The field must contain all upper case (A-Z) characters.

Also, remember that these keywords are not the only way to define "charsets". Any set of characters may be specified by placing the desired character set in quotes as the argument to this keyword.

Note that "charset" and "mapping" apply to each individual character that is entered into the form, while "translate_proc" and "check_proc" apply to the field as a whole and are only applied when the field is exited.

check_proc Field Statement:

Syntax:

```
     check_proc:  "‹Multics_pathname›";
     check_proc:  like ‹previously_defined_field›;
```

Example:

```
     check_proc:  "›udd›UserProj›UserName›form_utils_$check_this_one";
     check_proc:  "form_utils_$check_this_one";
     check_proc:  like date_field;
```

Description:

The "check_proc" field statement is used to define a procedure (program) that is called to validate the contents of the field upon exiting the field. The program is called with four parameters: the form_index, the field_name, the field_value, and a standard error code.

If the returned error code is non-zero, the user is not allowed to exit the field until the data is corrected. Currently, there is no means of communicating the nature or location of the error to the user directly, however, the "check_proc" is free to modify any attribute of <u>any</u> field, such as activating an error control field to inform the user of the nature of his error using any of the vform_ subroutines.

Notes:

The default is no "check_proc".

The "‹Multics_pathname›" must be a quoted string and may be either a relative or absolute pathname. If the pathname is a relative pathname (begins with a "‹"), the check procedure is searched for in the directory specified relative to the current working directory. If the pathname is absolute (begins with a "›"), the check procedure is searched for in the directory specified. If the pathname is simply an entry name (no "›" or "‹"'s), the check procedure is located using the vform search paths. See the add_search_paths command in the manual AG92, Multics Commands and Active Functions, for more information about the search path facility.

If both a "translate_proc" and a "check_proc" are defined for a particular field in the form, the "translate_proc" is called before the "check_proc".

It is recommended that a standard system error code be returned even though no action is currently taken to convey any information other than "beeping" the user's terminal. This is because it is planned that at some time in the future, a "status line" may be implemented as one of the screen lines that will display these messages in their text form. An error code called vf_et_$invalid_data whose message is "Invalid data for this field" has been provided for this purpose. For a discussion of creating customized error codes and a partial list of available error_table_ codes along with their text messages, see the manual AG91, Multics Programmer's Reference Manual, Section 7.

All parameters to the check_proc except for the "code" parameter are passed "by value" which means that when the check_proc returns, any modifications to parameters other than the "code" parameter will be ignored when the check_proc returns. If the programmer wishes to change the value of the field for which the check_proc was called, the programmer should call the vform_$set_value subroutine from within the check_proc.

The vform_ software calls the "check_proc" internally, so it should be written to accept parameters as if it were declared and called as follows:

```
     dcl  check_proc entry (fixed bin(35), char(*), char(*), char(*),
          fixed bin(35));

     call check_proc ((form_index), (form_name), (field_name),
          (field_value), code)
```

This is not to imply that the applications program needs to call the "check_proc" procedure. This information is provided to describe the interface used by the vform_ software to call the "check_proc" so that the applications programmer will know how to design the the check procedure. The "check_proc" should expect these parameters and handle them accordingly.

class Field Statement:

Syntax:

```
     classes:  <class_list>;
     class:    like <previously_defined_field>;
     classes:  like <previously_defined_field>,<class_list>;
```

Example:

```
     classes:  class_1,error_class,upper_class;
     class:    like first_field;
     classes:  like other_field,class_2,lower_class,middle_class;
```

Description:

The "class" field statement is used to define one or more classes to which the current field belongs. Operations can be performed on the fields in a class as a group, rather than performing the operation on each field separately.

Notes:

This feature can be quite useful for combining several functions into one form or may be used in situations that require different additional information depending upon the value of a specific field.

This feature may also be used to change the attributes or values of all the fields belonging to the specified class with one vform_ call.

column Field Statement:

Syntax:

```
     column:   ‹column_expression›;
     col:      ‹column_expression›;
```

Examples:

```
     col:      5;
     column:   data_field;
     column:   overlay other_field + 3;
     column:   like title_field;
     col:      like another_field - 2;
     col:      overlay data_field - 2;
```

Description:

The "column" field statement is used to define the column on the form (relative to the form origin).

The ‹column_expression› which describes the location of the field on the screen may be expressed in several forms. It may be:

- o An unsigned decimal integer specifying the exact location on the screen
- o Defined to be "like" or "overlay" with respect to a previously defined field in the form. The "like" and "overlay" keywords are synonymous.
- o Defined to begin in the column immediately following the end of a previously defined field.
- o Defined relative to the beginning or the end of a previously defined field using expressions of simple addition or subtraction.

See the "Notes" section for some examples that further explain the usage.

A field may also be defined relative to the end of a previously defined field exactly as described above, but leaving out the "like" or "overlay" token. See the Notes section for more information.

Notes:

Some examples follow to attempt to clarify the use of the various options to the column statement.

```
[1]   field:          old_field;
       column:        5;
       row:           2;
           .
           .
           .
     field:           new_field;
       column:        like old_field;
       row:           3;
```

Example [1] will result in new_field existing in the same column (column 5) as old_field.

```
[2]   field:          old_field;
       col:           5;
       row:           2;
           .
           .
           .
     field:           new_field;
       col:           overlay old_field + 2;
       row:           3;
```

Example [2] will result in new_field being two columns over from the beginning of old_field (column 7);

```
[3]  field:           old_field;
       col:           5;
       length:        3;
           .
           .
           .
     field:           new_field;
       col:           old_field + 1;
```

Example [3] will result in new_field being located in screen location 1 column beyond the end of old_field (column 9);

length Field Statement: Syntax:

```
     length:  ‹unsigned_decimal_integer›;
     len:  like ‹previously_defined_field›;
```

Examples:

```
     len:  12;
     length:  like data_field;
```

Description:

The "length" field statement determines the length of the field that is being defined.

Notes:

A "length" or a "value" field statement is required for each field.

If a "value" field statement is specified, there is an implicit "length" field statement whose value is the length of the string specified in the "value" field statement. This default can be overridden with an explicit "length" field statement.

mapping Field Statement:

Syntax:

```
     mapping:  ‹quoted_string_or_keyword›,‹quoted_string_or_keyword›;
     map:  ‹quoted_string_or_keyword›,‹quoted_string_or_keyword›;
```

Examples:

```
     map:      upper,lower;
     mapping:  "xyz","ABC";
```

Description:

The "mapping" field statement is used to indicate that one set of characters is "mapped" or translated to another set of characters as they are entered. This capability allows the implementation of upper case only fields, etc.

Notes:

The "mapping" field statement is optional, and if omitted, no mapping is done. The keywords provided are "upper", and "lower", thus the following "mapping" field statement would indicate that all lower case characters are mapped (or translated) to upper case characters.

```
     mapping:  lower, upper;
     mapping:  "abcdefghijklmnopqrstuvwxyz","ABCDEFGHIJKLMNOPQRSTUVWXYZ"
```

Note that "charset" and "mapping" apply to each individual character that is entered into the form, while "translate_proc" and "check_proc" apply to the field as a whole and are only applied when the field is exited.

next_field Field Statement:

Syntax:

```
     next_field:  ‹field_name›;
     next:  ‹field_name›;
```

Examples:

```
     next_field:  data_field;
     next:  data_field;
```

Description:

The "next_field" field statement provides a mechanism for defining "connected" fields. A set of connected fields might be used to define a group of fields on a form that, together, make up a piece of text. It may also be used to hold data that is made up of more than one line on the form. Connected fields are special in that deleting a character from the first, causes a character to be moved up from the second, and so on. Inserting characters in a connected field causes characters which are pushed off the end of the first field to be inserted in the beginning of the second, and so on.

Notes:

It is often necessary to place a block of text on a form, and "next_field" is the facility that allows a user to enter and edit that data. The capabilities provided are primitive, but should allow a user to accomplish most editing necessary for a forms application.

The last field in a chain of fields with "next" fields may have the "^overflow" attribute. If this is the case, and the user attempts to insert data that would cause data to be pushed off the screen, the sub_error_ condition is signaled as described in the "attributes" field statement above.

row Field Statement:

Syntax:

```
     row:       <unsigned_decimal_integer>;
     row:       like <previously_defined_field>;
     row:       like <previously_defined_field> ± <unsigned_decimal_integ
```

Example:

```
     row:       4;
     row:       like other_field;
     row:       like other_field + 2;
     row:       like field_x - 1;
```

Description:

The "row" field statement defines the row on the screen (relative to the origin of the form) where the field is to be placed. The row may be specified "like" that of another field, and it will be on the same row. The row may be specified "like" that of another field, but with an additional offset, in which case, the field is defined relative to the field it is "like".

A "row" statement is required for each field, unless it is "like" another field in which case the row is the same as that field.

_translate_proc_ Field Statement:

Since a check_proc may now modify its "field_value" parameter, translate_proc's are obsolete.

value Field Statement:

Syntax:

```
     value:    <quoted_string>;
     val:      like <previously_defined_field>;
```

Example:

```
     value:    "Part Number";
     value:    like data_field1;
     val:      "Purchase Order"
```

Description:

The "value" field statement gives a particular field its initial value. This statement is generally used to give protected (or title) fields their values, or to give unprotected (or data) fields their initial values.

Notes:

A "value" or a "length" field statement is required for each field.

Specifying a value for a field implies a length for the current field equal to the length of the string specified in the "value" field statement. This length may be overridden with an explicit "length" field statement.

Section 4

Subroutines

This is a description of the various subroutine entry points available to the forms programmer. These subroutine entry points allow the programmer to open, close, and manipulate the various properties of a form.

A simpler method for calling each of the subroutines described in this section is available when a call to sub_err_ is appropriate when an error occurs. The call to sub_err_ prints the error message and places the user at a new command level. See Appendix D for details.

Entry: vform_$assign_values

This entry point is designed to allow a programmer to assign values to the fields of a form. It is passed an array of the names of the fields that are to be given values and an array of the values to be assigned.

Usage

```
     dcl  vform_$assign_values entry (fixed bin(35),
          (*) char(*) aligned, (*) char(*) aligned, fixed bin(35),
          fixed bin(35));

     call vform_$assign_values (form_index, name_array, data_array,
          error_index, code);
```

where:

1. form_index (Input)
   is a valid form index obtained by opening a form with the subroutine entry point vform_$open_form.

2. name_array (Input)
   is an array of the names of the fields which are to have values assigned to them. This array must be aligned on a word boundary and the lower bound of the name array must be equal to the lower bound of the data_array. The names in name_array may be in any order and any number of fields may be assigned with one call. It is recommended that the dimension of the name_array and data_array be identical. The upper bound of the data_array may be larger than the upper bound of the name_array but if so, these elements will be ignored

3. data_array (Input)
   is an array containing the values of fields to be assigned whose names are contained in the corresponding element of the name_array. See name_array above for a description of the alignment and dimension of the arrays.

4. error_index (Output)
   is valid only if the code (see below) is not zero. If an error occurred while assigning a value to a field in the name_array then the error index will be the index into the name_array of the field name that caused the error. See Notes below.

5. code (Output)
   is a standard status code.

Notes:

The arrays which are passed to this subroutine entry point must be aligned so that the vform_ package will be compatible with Fortran 77 and Cobol. Both Fortran 77 and Cobol arrays are aligned on word boundaries and cannot be forced otherwise. PL/1 arrays are by default unaligned, so PL/1 programs using this subroutine entry point must force the alignment of the arrays which are used as parameters to this subroutine entry point.

The error_index parameter is used to indicate to the programmer which of the field names in the field array caused the error to occur. Thus, this parameter has no meaning when code is equal to zero. For example, if the name_array has 10 elements, but the name of the field specified in name_array(4) is not a valid name of a field of the form, then code will have a value of vf_et_$field_not_found_in_form, and error_index will have a value of 4.

If error_index is zero, code is not necessarily zero. There may be other errors not related to a specific field name specified in the name_array.

If any error is detected, processing of the arrays stops, and no further fields are assigned.

The command vf_create_include_file may be used to create an include file which defines data structures suitable for use in conjunction with this subroutine entry point. This command is documented in section 5.

Entry: vform_$class_blank_field_count

This entry point returns the number of fields in a given class whose value is blank.

Usage

```
     dcl  vform_$class_blank_field_count entry (fixed bin(35),
          char(*), fixed bin(35), fixed bin(35));

     call vform_$class_blank_field_count entry (form_index,
          class_name, blank_field_count, code);
```

where:

1. form_index (Input)
   is a valid form index obtained by opening a form with the subroutine entry point vform_$open.

2. class_name (Input)
   is the name of the class whose blank field count is desired.

3. blank_field_count (Input)
   is the number of fields in the given class whose value is blank.

4. code (Output)
   is a standard status code.

Notes:

The programmer should be aware that ALL fields in a class are checked, not just the active or non-protected ones.

Entry: vform_$class_is_all_blank

This entry point is used to determine if all the fields in a class have values of blanks only.

Usage

```
     dcl  vform_$class_is_all_blank entry (fixed bin(35), char(*),
          fixed bin(35), fixed bin(35));

     call vform_$class_is_all_blank (form_index, class_name, is_blank,
          code);
```

where:

1. form_index (Input)
   is a valid form index obtained by opening a form with the subroutine entry point vform_$open.

2. class_name (Input)
   is the name of the class whose values are to be checked.

3. is_blank (Output)
   has the value of 1 if all of the fields in the class are blank and has the value of 0 if at least one field in the class is non-blank.

4. code   (Output)
   is a standard status code.

Notes:

The programmer should be aware that ALL fields in a class are checked, not just the active or non-protected ones.

Entry: vform_$class_is_all_non_blank

This entry point is used to determine of all of the fields in a given class have non-blank values.

Usage

```
     dcl  vform_$class_is_all_non_blank entry (fixed bin(35), char(*),
          fixed bin(35), fixed bin(35));

     call vform_$class_is_all_non_blank (form_index, class_name,
          is_non_blank, code);
```

where:

1. form_index (Input)
   is a valid form index obtained by opening a form with the subroutine entry point vform_$open.

2. class_name (Input)
   is the name of the class whose values are to be tested.

3. is_non_blank (io)
   has a value of 1 if all fields in the given class have non-blank values and has a value of 0 if at least one field in the given class has a blank value.

4. code (Output)
   is a standard status code.

Notes:

This entry point can be used to determine if all values in a given class have been filled in, that is, it can be used to implement "required fields".

The programmer should be aware that ALL fields in a class are checked, not just the active or non-protected ones.

Entry: vform_$class_non_blank_field_count

This entry point is used to determine how many fields in a given class have non-blank values.

Usage

```
dcl  vform_$class_non_blank_field_count entry (fixed bin(35),
          char(*), fixed bin(35), fixed bin(35));

call vform_$class_non_blank_field_count (form_index, class_name,
          non_blank_field_count, code);
```

where:

1. form_index (Input)
   is a valid form index obtained by opening a form with the subroutine entry point vform_$open.

2. class_name (Input)
   is the name of the class whose non-blank field count is to be returned for.

3. non_blank_field_count (Output)
   is the number of fields in the given class whose value is non-blank.

4. code (Output)
   is a standard status code.

Notes:

The programmer should be aware that ALL fields in a class are checked, not just the active or non-protected ones.

Entry: vform_$clear_screen

This entry point clears the screen of the terminal. It should not be used from a check_proc or an exit_proc unless it also restores the image of the screen using vform_$display_form.

Usage

```
     dcl  vform_$clear_screen entry (fixed bin(35), fixed bin(35));

     call vform_$clear_screen (form_index, code);
```

where:

1. form_index (Input)
   is a valid form index obtained by opening a form with the subroutine entry point vform_$open_form.

2. code (Output)
   is a standard status code.

Notes:

The form_index is required for this entry point so that the vform_ software can be sure that the terminal type information has been initialized in the process.

This procedure does not communicate the fact that the screen has been altered to other form software, so it should not be used from a check_proc or exit_proc unless the check_proc or exit_proc is prepared to restore the screen.

Entry: vform_$clear_unprotected_fields

This entry point clears all of the unprotected fields in the specified form. Clearing a field is equivalent to its value to blanks.

Usage

```
     dcl  vform_$clear_unprotected_fields entry (fixed bin(35),
          fixed bin(35));

     call vform_$clear_unprotected_fields (form_index, code);
```

where:

1. form_index (Input)
   is a valid form index obtained by opening a form with the subroutine entry point vform_$open_form.

2. code
   is a standard status code.

Notes:

Although the routine is quite effective, it is more efficient to define a "class" of fields which contains all of the unprotected fields of the form and use the vform_$set_class_value subroutine entry point to set the value of all fields in the class to blanks.

Entry: vform_$close_debug_file

This entry point closes the debugging file attached with the vform_$debug_atd entry point.

Usage

```
     dcl  vform_$close_debug_file entry ();

     call vform_$close_debug_file ();
```

Notes:

The debugging feature should be disabled using the vform_$debug_off subroutine entry point before the debugging file is closed to prevent the debugging feature from displaying information on the user's terminal.

See the entries for vform_$debug_atd, vform_$debug_on, and vform_$debug_off for more information.

Entry: vform_$close_form

This entry point closes a form that has been previously opened by the vform_$open_form subroutine entry point. Closing a form invalidates the form index and frees a considerable amount of storage used for each form. Since the amount of temporary storage available to a user is finite, it is highly recommended that forms be closed after the application is done with it, and that cleanup handlers be used to insure that a form is closed if a program is interrupted and its stack frame released. Again, closing a form frees a considerable amount of storage.

Usage

```
     dcl  vform_$close_form entry (fixed bin(35), fixed bin(35));

     call vform_$close_form (form_index, code);
```

where:

1. form_index (Input)
   is a valid form index obtained by opening a form with the subroutine entry point vform_$open_form.

2. code   (Output)
   is a standard status code.

Notes:

If the form is successfully closed, the form index will be set to zero.

It is not possible to over-emphasize the importance of closing forms when the program has finished with them. Not closing the forms may cause serious debugging problems as open forms take up considerable space in the process directory. Not closing the forms may cause fatal process errors when too many forms are opened or when the user runs a program too many times. The moral of the story is, close all forms when done, and establish cleanup handlers to close forms if/when a program terminates unexpectedly.

Entry: vform_$debug_atd

This entry point is used to attach the vform_debug_ I/O switch with the specified attach description. This entry point is used in conjunction with the vform_$debug_on and vform_$debug_off entry points to produce a tracing of what is happening with the various forms subroutines.

Usage

```
     dcl  vform_$debug_atd entry (char(*));

     call vform_$debug_atd (attach_desc);

                    -or-

     call vform_$debug_atd ("vfile_ debug_output_file");
```

where:

1. attach_desc (Input)
   is an attach description acceptable to the iox_$attach_name or iox_$attach_ptr subroutines.

Notes:

See the Multics Programmers' Manual Reference Guide (AG91) Section 5 for a discussion of I/O switches. See the Multics Programmers' Manual Subroutines (AG93) Section 2 for a discussion of the iox_$attach_name and iox_$attach_ptr subroutines.

The vform_$debug_atd entry point simply sets up the vform_debug_ I/O switch. The vform_$debug_on subroutine must be used to start the tracing, and the vform_$debug_off subroutine will stop the tracing.

The applications program is welcome to output its own tracing information on the vform debug iocb, but it is suggested that the applications program not modify the attributes of the iocb. The iocb is called vfs_$debug_iocb and should be declared:

```
     dcl  vfs_$debug_iocb pointer external static;
```

Entry: vform_$debug_off

This entry point turns off the tracing started with vform_$debug_on. It does not close the debug output file.

Usage

```
     dcl  vform_$debug_off entry ();

     call vform_$debug_off ();
```

Notes:

The vform_$debug_off entry point is used to stop the tracing information from being written to the debugging iocb. This is so tracing may be stopped and later restarted. It does not close the debugging file. The vform_$close_debug_file must be used to close the file.

See also the vform_$debug_on, vform_$debug_atd, and vform_$close_debug_file subroutine entry points for additional information.

Entry: vform_$debug_on

This entry point enables (or reenables) various forms of tracing to be sent to the vform debugging iocb. See the vform_$debug_atd subroutine entry point for more information.

Usage

```
     dcl  vform_$debug_on entry ();

     call vform_$debug_on ();
```

Notes:

The tracing that is enabled by this entry point is quite extensive, and could take up quite a bit of disk space.

This entry point is provided mostly for the use of developers of the vform_ package, but may be of some use to the applications program. Most of the events that happen with respect to the forms software are logged and a great deal of other information is recorded. However, it should be noted that all events are not recorded in the debugging log. A more complete and/or selective logging may be provided as a future enhancement.

It is recommended that the vform_$debug_atd subroutine entry point be called before tracing is started to prevent information from being sent to the user's terminal. The I/O switch user_output is used by default.

Entry: vform_$disable_exit_proc

This entry point is used to disable the calling of the "exit_proc" for a form when the form is exited. This is particularly useful for the case when the same form is used for input and retrieval. In this case, some data fields may be protected and contain invalid data. The "exit_proc" would find the invalid data and would prevent the user from exiting the form normally.

Usage

```
     dcl  vform_$disable_exit_proc entry (fixed bin(35),
          fixed bin(35));

     call vform_$disable_exit_proc (form_index, code);
```

where:

1. form_index (Input)
   is a valid form index obtained by opening a form with the subroutine entry point vform_$open_form.

2. code   (Output)
   is a standard status code.

Notes:

The calling of the exit_proc may be enabled with a call to vform_$enable_exit_proc.

Entry: vform_$display_form

This entry point is used to display a form on the screen. This operation should be done before a call to vform_$read_form. It causes a complete redisplay of the screen and should only be done when necessary such as the first time a form is displayed on the screen or when some output external to the forms package may have damaged the screen contents as this operation is both processor and real time consuming. It is not necessary to re-display a form after each modification This may be useful to display a form that contains information and does not require any user interaction, or to insure that the screen is in a known state.

Usage

```
     dcl  vform_$display_form entry (fixed bin(35), fixed bin(35));

     call vform_$display_form (form_index, code);
```

where:

1. form_index (Input)
   is a valid form index obtained by opening a form with the subroutine entry point vform_$open_form.

2. code (Output)
   is a standard status code.

Notes:

The vform_$display_form routine may be used in cases where the programmer simply wants to display the form on the screen, such as after a different form has been on the screen or when the program has caused output to the screen other than output from the forms package.

Entry: vform_$enable_exit_proc

This entry point is used to enable the calling of the "exit_proc" for a form after the feature has been disabled with vform_$disable_exit_proc.

Usage

```
     dcl  vform_$enable_exit_proc entry (fixed bin(35),
          fixed bin(35));

     call vform_$enable_exit_proc (form_index, code);
```

where:

1. form_index (Input)
   is a valid form index obtained by opening a form with the subroutine entry point vform_$open_form.

2. code (Output)
   is a standard status code.

Notes:

The calling of an exit_proc is enabled by default, but can be disabled with the vform_$disable_exit_proc subroutine entry point.

Entry: vform_$extract_class_values

This entry point is used to extract the values for a class of fields into the given data array. Since the order in which the data is returned is not defined, this entry point is of little use.

Usage

```
     dcl  vform_$extract_class_values entry (fixed bin(35), char(*),
          (*) char(*), fixed bin(35), fixed bin(35));

     call vform_$extract_class_values (form_index, class_name,
          data_array, nused, code);
```

where:

1. form_index (Input)
   is a valid form index obtained by opening a form with the subroutine entry point vform_$open_form.

2. class_name (Input)
   is the name of the class whose values are to be extracted.

3. data_array (Output)
   is an aligned array whose dimension should be large enough to accommodate all of the fields in the given class. The array must be word aligned.

4. nused (Output)
   is the number of entries in the array that were used. This allows the programmer to pass in an array larger than is necessary with no loss of efficiency since those elements of the array that are not needed are not used.

5. code (Output)
   is a standard status code.

Notes:

For a discussion of the required alignment of the array parameter, see the description for the vform_$extract_values subroutine entry point.

The order in which data is returned is not defined, and the only real use for this entry point would be to test to see if all of the fields in a particular class have the same value.

Entry: vform_$extract_values

This entry point is designed to allow the programmer to retrieve the data entered into the form. It is passed an array of the names of the fields that are to be returned, and returns an array of the data corresponding to the name array.

Usage

```
     dcl  vform_$extract_values entry (fixed bin(35),
          (*) char(*) aligned, (*) char(*) aligned, fixed bin(35),
          fixed bin(35));

     call vform_$extract_values (form_index, name_array, data_array,
          error_index, code);
```

where:

1. form_index (Input)
   is a valid form index obtained by opening a form with the subroutine entry point vform_$open_form.

2. name_array (Input)
   is an array of the names of the fields to be returned. This array must be aligned on a word boundary and the lower bound (lbound) of the array must be equal to the lower bound of the data_array. The names in name_array may be in any order and any number of fields may be returned with one call. It is recommended that the dimension of the name_array and the data_array be identical. The upper bound (hbound) of data_array may be larger that the upper bound of name_array, but these elements will be ignored, and thus their values will be undefined.

3. data_array (Output)
   is an array that the data in the field specified by the corresponding item of the name_array is returned into. This array must be aligned on a word boundary, and the lower bound (lbound) of the array must be equal to the lower bound (lbound) of the name_array. The upper bound (hbound) of the data_array must be greater than or equal to the upper bound (hbound) of the name_array. It is recommended that the dimension of the name_array and the data_array be identical.

4. error_index (Output)
   is valid only if the code (see below) is not equal to zero. If an error occurred while extracting one of the values for a field specified in the name_array (see above), then the error_index will be the index into the array of the field that caused the error. See Notes below.

5. code (Output)

is a standard status code.

Notes:

The arrays that are passed to this routine must be aligned so that the vform_ package will be compatible with Fortran and Cobol. Both Fortran 77 and Cobol arrays are aligned on word boundaries, and cannot be forced otherwise. PL/1 arrays can be made to be aligned or unaligned, and unaligned is the default, therefore, PL/1 programs must force alignment of the arrays used in this entry point.

The parameter error_index is used to indicate to the programmer which of the field names in the field array caused the error to occur. Thus, this parameter has no meaning when code is equal to zero. For example, if the name_array has 10 elements, but the name of the field specified in name_array(4) is not a valid name of a field of the form, then code will have a value of vf_et_$field_not_found_in_form, and error_index will have a value of 4. The value of error_index being zero is not an indication that no error occurred. There may be other errors not relating to a specific field name specified in the name_array.

The command vf_create_include_file may be used to create an include file which defines data structures suitable for use in conjunction with this subroutine entry point. This command is documented in section 5.

Entry: vform_$get_attributes

This entry returns the attributes of a given field.

Usage

```
     dcl  vform_$get_attributes entry (fixed bin(35), char(*),
          char(*), fixed bin(35));

     call vform_$get_attributes (form_index, field_name, attributes,
          code);
```

where:

1. form_index (Input)
   is a valid form index obtained by opening a form with the subroutine entry point vform_$open_form.

2. field_name (Input)
   is the name of the field whose attributes are to be retrieved.

3. attributes (Output)
   is the current attributes of the specified field. This string should be 256 characters long to accommodate the longest mode string ever anticipated.

4. code (Output)
   is a standard status code.

Notes:

The attributes returned may be any of the names described in the attributes field statement in Section 3. Their order and which abbreviations are returned is not defined. Use the vform_$test_attributes to test to see if a field has a given set of attributes.

The string returned will be acceptable to as input to vform_$set_attributes.

Entry: vform_$get_class_modified_flag

This entry point is used to determine whether or not the value of any field in the specified class of fields was modified during the last call to vform_$read_form or vform_$update_form.

Usage

```
     dcl  vform_$get_class_modified_flag entry (fixed bin(35),
          char(*), fixed bin(35), fixed bin(35));

     call vform_$get_class_modified_flag (form_index, class_name,
          modified, code);
```

where:

1. form_index (Input)
   is a valid form index obtained by opening a form with the subroutine entry point vform_$open_form.

2. class_name (Input)
   is the name of the class whose modified flags are to be checked.

3. modified (Output)
   is an integer number whose value will be 0 if no fields in the specified class were modified, and 1 if any of the fields in the class have been modified.

4. code (Output)
   is a standard status code.

Notes:

The modified parameter is a fixed bin(35) number, but its value will be either 0 or 1 only.

The internal modified flag is automatically reset with each call to vform_$read_form or vform_$update_form.

Entry: vform_$get_field_modified_flag

This entry point is used to determine whether or not the value of a field was modified since the last call to vform_$read_form or vform_$update_form.

Usage

```
     dcl  vform_$get_field_modified_flag entry (fixed bin(35),
          char(*), fixed bin(35), fixed bin(35));

     call vform_$get_field_modified_flag (form_index, field_name,
          modified, code);
```

where:

1. form_index (Input)
   is a valid form index obtained by opening a form with the subroutine entry point vform_$open_form.

2. field_name (Input)
   is the name of the field whose modified flag is to be checked.

3. modified (Output)
   is an integer number whose value will be 0 if the field has not been modified and 1 if any of the field has been modified.

4. code (Output)
   is a standard status code.

Notes:

The modified parameter is a fixed bin(35) number, but its value will be either 0 or 1 only.

The internal modified flag is automatically reset with each call to vform_$read_form or vform_$update_form.

Entry: vform_$get_modified_flag

This entry point returns information telling the caller whether any of the data in the fields of the form has been modified since the last call to vform_$read_form or vform_$update_form.

Usage

```
     dcl  vform_$get_modified_flag entry (fixed bin(35),
          fixed bin(35), fixed bin(35));

     call vform_$get_modified_flag (form_index, modified, code);
```

where:

1. form_index (Input)
   is a valid form index obtained by opening a form with the subroutine entry point vform_$open_form.

2. modified (Output)
   is an integer whose value is 0 if no data in the fields of the form has been modified, and 1 if any of data in the fields of the form has been modified.

3. code (Output)
   is a standard status code.

Notes:

The modified parameter is a fixed bin(35) number, but its value will be either 0 or 1.

The modified flag is an indication of the data in a field changing, not its attributes, thus if a field changes from blinking to ^blinking, its data modified flag will not change.

The internal modified flag is automatically reset by each call to vform_$read_form or vform_$update_form. The modified flags may also be reset by a call to vform_$reset_modified_flag or vform_$reset_class_modified_flag.

Entry: vform_$get_value

This entry point returns the current value of a specified field.

Usage

```
     dcl  vform_$get_value entry (fixed bin(35), char(*), char(*),
          fixed bin(35));

     call vform_$get_value (form_index, field_name, value, code);
```

where:

1. form_index (Input)
   is a valid form index obtained by opening a form with the subroutine entry point vform_$open_form.

2. field_name (Input)
   is the name of the field whose value is to be retrieved.

3. value (Output)
   is the value of the specified field.

4. code (Output)
   is a standard status code.

Entry: vform_$open_form

This entry point is used to perform the necessary initialization of a form. It returns a form_index to be used by all other subroutine entry points dealing with forms.

Usage

```
     dcl  vform_$open_form entry (char(*), fixed bin(35), fixed
          bin(35));

     call vform_$open_form (form_path, form_index, code);
```

where:

1. form_path (Input)
   is the pathname of a converted form segment. A converted form segment is created by the cv_form routine. Archive component pathnames and either absolute or relative pathnames are allowed.

2. form_index (Output)
   is a number unique to the current opening of the form. This index is used by the other vform_ subroutine entry points to identify this opening of this form.

3. code   (Output)
   is a standard status code.

Notes:

A particular form may be opened several distinct times by one or more applications. Each opening will have its own copy of the initial data from the form segment and each opening is independent of the others.

If the form_path is an entry_name (contains no "<" or ">" characters), then the "vform" search paths will be used to locate the form. See the add_search_paths command in the manual AG92, Multics Commands and Active Functions for a discussion of the search path facility.

Entry: vform_$position_cursor

This entry point allows the programmer to specify the initial field (and column within the field) to place the cursor in when a call to vform_$read_form, vform_$read_transmit_form, or vform_$update_form is made. The field specified must be active and ^protected when the call to vform_$read_form, vform_$read_transmit_form, or vform_$update_form is made.

Usage

```
     dcl  vform_$position_cursor entry (fixed bin(35), char(*),
          fixed bin(35), fixed bin(35));

     call vform_$position_cursor (form_index, field_name, column,
          code);
```

where:

1. form_index (Input)
   is a valid form index obtained by opening a form with the subroutine entry point vform_$open.

2. field_name (Input)
   is the name of the active, unprotected field that the cursor is to be placed in.

3. column (Input)
   is the column (relative to the beginning of the field) that the cursor is to be placed in.

4. code (Output)
   is a standard status code.

Notes:

The "column" specified is relative to the beginning of the field, not to the edge of the screen.

The field specified must be active and ^protected when a call to vform_$read_form or vform_$read_transmit_form is made or the cursor will be automatically placed to the first field that is active and ^protected.

The cursor position specified is stored internally to the Virtual Forms software and is remembered until the next call to vform_$read_form, vform_$read_transmit_form, or vform_$update_form.

Calls to vform_$read_form or vform_$read_transmit_form position the cursor to the specified field once, then forget the specified position i.e. the cursor is only placed in the specified field once and then forgotten. Calls to vform_$update_form honor the cursor position specified, but do not erase it which basically passes it through to the next call to vform_$read_form or vform_$read_transmit_form.

Entry: vform_$read_form

This entry point processes the user's input from a form already on the screen until the time that the proper key sequence or function key is sent to exit (transmit) or abort the form. This routine assumes that the form specified by the form_index is currently displayed on the screen, that is, it does not display the form for you.

Usage

```
     dcl  vform_$read_form entry (fixed bin(35), fixed bin(35));

     call vform_$read_form (form_index, code);
```

where:

1. form_index (Input)
   is a valid form index obtained by opening a form with the subroutine entry point vform_$open_form.

2. code (Output)
   is a standard status code.

   The set of possible error codes includes, but is not limited to:

   vf_et_$form_aborted
   when the user aborts the form

Notes:

It is necessary that the form be correctly displayed on the screen when this routine is called. This subroutine entry point updates the screen with any changes since the last call to vform_$display_form, vform_$read_form, or vform_$update_form, but it does not display the form on the screen.

The vform_$read_form entry point resets the modified flag for all the fields in the form.

Entry: vform_$read_transmit_form

This entry point is like the vform_$read_form except that the only allowed operations are "TRANSMIT FORM" and "ABORT FORM". No data may be entered when using this entry point. This entry point should be used when the program wishes to display the data of a form to the user when there are no unprotected fields in the form.

Usage

```
     dcl  vform_$read_transmit_form entry (fixed bin(35), fixed
          bin(35));

     call vform_$read_transmit_form (form_index, code);
```

where:

1. form_index (Input)
   is a valid form index obtained by opening a form with the subroutine entry point vform_$open.

2. code (Output)
   is a standard status code.

Notes:

This entry point would be used for situations when all the data on a form is protected and the programs wishes the user to "view" the data without being able to modify any of it. This could be used to implement a "browse" mode or in read-only retrieval mode.

Entry: vform_$reset_class_modified_flag

This entry point resets the data modified flag for each field in the given class of fields.

Usage

```
     dcl  vform_$reset_class_modified_flag entry (fixed bin(35),
          char(*), fixed bin(35));

     call vform_$reset_class_modified_flag (form_index, class_name,
          code);
```

where:

1. form_index (Input)
   is a valid form index obtained by opening a form with the subroutine entry point vform_$open_form.

2. class_name (Input)
   is the name of the class of fields for which the data modified flags will be reset.

3. code   (Output)
   is a standard status code.

Notes:

The modified parameter is a fixed bin(35) integer whose value will be 0 or 1.

Note that the modified flag indicates whether or not the data in the field has been modified, not whether the attributes of the field have been modified. Thus, changing a field from "inverse" to "^inverse", does not affect this flag.

The internal modified flag is reset automatically by the vform_$read_form and the vform_$update_form entry points.

Entry: vform_$reset_form

This entry point resets the specified form to its initial state, that is, the state at the time that the form was opened.

Usage

```
     dcl  vform_$reset_form entry (fixed bin(35), fixed bin(35));

     call vform_$reset_form (form_index, code);
```

where:

1. form_index (Input)
   is a valid form index obtained by opening a form with the subroutine entry point vform_$open.

2. code (Output)
   is a standard status code.

Notes:

This entry point gets its information from internally maintained information. It does not go back to the converted form segment for its information, thus reconverting the form then using the vform_$reset_form entry point will not reflect these changes. To do this, the form must be closed and re-opened.

Entry: vform_$reset_modified_flag

This entry point resets the data modified flag for all fields in the form.

Usage

```
     dcl  vform_$reset_modified_flag entry (fixed bin(35),
          fixed bin(35));

     call vform_$reset_modified_flag (form_index, code);
```

where:

1. form_index (Input)
   is a valid form index obtained by opening a form with the subroutine entry point vform_$open_form.

2. code (Output)
   is a standard status code.

Notes:

Note that the modified flag indicates whether or not the data in the field has been modified, not whether the attributes of the field have been modified. Thus, changing a field from "inverse" to "^inverse", does not affect this flag.

The internal modified flag is reset automatically by the vform_$read_form and the vform_$update_form subroutine entry points.

Entry: vform_$set_attributes

This entry point is used to change the attributes of a field in a specified form.

Usage

```
     dcl  vform_$set_attributes entry (fixed bin(35), char(*),
          char(*), fixed bin(35));

     call vform_$set_attributes (form_index, field_name,
          new_attributes, code);

                    -or-

     call vform_$set_attributes (form_index, "input_date",
          "active,inv,^prot", code);
```

where:

1. form_index (Input)
   is a valid form index obtained by opening a form with the subroutine entry point vform_$open_form.

2. field_name (Input)
   is the name of the field whose attributes are to be changed.

3. new_attributes (Input)
   is a string describing the new attributes that the field is to take on. This string has the same format as the <attribute_list> described in the attribute field statement in Section 3.

4. code   (Output)
   is a standard status code.

Entry: vform_$set_class_attributes

This entry point is used to change the attributes of a class of fields in a specified form.

Usage

```
     dcl  vform_$set_class_attributes entry (fixed bin(35), char(*),
          char(*), fixed bin(35));

     call vform_$set_class_attributes (form_index, class_name,
          new_attributes, code);

                         -or-

     call vform_$set_class_attributes (form_index, "input_date",
          "active,inv,^prot", code);
```

where:

1. form_index (Input)
   is a valid form index obtained by opening a form with the subroutine entry point vform_$open_form.

2. class_name (Input)
   is the name of the class of fields to be modified.

3. new_attributes (Input)
   is a string describing the new attributes that the class of fields is to take on. This string has the same format as the ‹attribute_list› described in the attribute field statement in Section 3. Only those attributes specified are modified. All other attributes remain unchanged.

4. code (Output)
   is a standard status code.

Entry: vform_$set_class_value

This entry point is used to change the value of a class of fields. Fields may be modified regardless of their attributes, that is, the data in a protected field may be modified with a call to this program.

Usage

```
     dcl  vform_$set_class_value entry (fixed bin(35), char(*),
          char(*), fixed bin(35));

     call vform_$set_class_value (form_index, class_name, new_value,
          code);
```

where:

1. form_index (Input)
   is a valid form index obtained by opening a form with the subroutine entry point vform_$open_form.

2. class_name (Input)
   is the name of the class whose value is to be changed.

3. new_value (Input)
   is the new value that the class of fields is to receive.

4. code (Output)
   is a standard status code.

Entry: vform_$set_value

This entry point is used to change the value of a field. Fields may be modified regardless of their attributes, that is, the data in a protected field may be modified with a call to this program.

Usage

```
     dcl  vform_$set_value entry (fixed bin(35), char(*), char(*),
               fixed bin(35));

     call vform_$set_value (form_index, field_name, new_value, code);

                         -or-

     call vform_$set_value (form_index, "data_title", "Enter data:",
               code);
```

where:

1. form_index (Input)
   is a valid form index obtained by opening a form with the subroutine entry point vform_$open_form.

2. field_name (Input)
   is the name of the field whose value is to be modified or obtained.

3. new_value (Input)
   is the new value to be given to the field.

4. code (Output)
   is a standard status code.

Entry: vform_$test_attributes

This entry point is used to test a field to see if that field has the given set of attributes.

Usage

```
     dcl  vform_$test_attributes entry (fixed bin(35), char(*),
          char(*), fixed bin(35), fixed bin(35));

     call vform_$set_attributes (form_index, field_name, attributes,
          return_value, code);

                    -or-

     call vform_$set_attributes (form_index, "input_date",
          "active,inv,^prot", return_value, code);
```

where:

1. form_index (Input)
   is a valid form index obtained by opening a form with the subroutine entry point vform_$open_form.

2. field_name (Input)
   is the name of the field whose attributes are to be changed.

3. attributes (Input)
   is a string describing the attributes to be tested for in the current field. This string has the same format as the ‹attribute_list› described in the attribute field statement in Section 3.

4. return_value (Output)
   is an integer whose value is 1 if the field has all of the attributes specified and 0 otherwise.

5. code (Output)
   is a standard status code.

Notes:

The value of return_value is undefined if the error code is non-zero.

Entry: vform_$test_class_attributes

This entry point is used to test all of the fields in a class to see if they have a given set of attributes.

Usage

```
     dcl  vform_$test_class_attributes entry (fixed bin(35), char(*),
          char(*), fixed bin(35), fixed bin(35));

     call vform_$test_class_attributes (form_index, class_name,
          attributes, return_value, code);

                    -or-

     call vform_$set_class_attributes (form_index, "input_date",
          "active,inv,^prot", return_value, code);
```

where:

1. form_index (Input)
   is a valid form index obtained by opening a form with the subroutine entry point vform_$open_form.

2. class_name (Input)
   is the name of the class of fields to be modified.

3. attributes (Input)
   is a string defining the list of attributes to be tested for in the class. This string has the same format as the ‹attribute_list› described in the attribute field statement in Section 3. Only those attributes specified are tested for.

4. return_value (Output)
   is an integer whose value is 1 if all of the fields in the specified class have all of the attributes as specified in the attributes list. Its value is 0 otherwise.

5. code (Output)
   is a standard status code.

Notes:

The value of return_value is undefined if code is non-zero.

Entry: vform_$update_form

This entry point forces a screen update to having without passing control to the user.

Usage

```
     dcl  vform_$update_form entry (fixed bin(35), fixed bin(35));

     call vform_$update_form (form_index, code);
```

where:

1. form_index (Input)
   is a valid form index obtained by opening a form with the subroutine entry point vform_$open.

2. code (Output)
   is a standard status code.

Notes:

This entry point is useful when it is desirable to convey some information to the user, without passing control to the user such as is done with vform_$read_form. An example of this may be to let the user know that the program is alive and well when the program is doing a long data base retrieve. Often the programmer may put a message in the form saying "Retrieving..." while the program is doing this to prevent the user form hitting keys with the feeling that the program has either died or is waiting for them to do something. This entry point allows the programmer to set the value of that field, then call vform_$update_form to make the changes appear on the screen immediately.

Calling this entry point also resets the modified flags for fields in the form.

Section 5

Form Related Commands

Name: cv_form

Syntax

cv_form path {-control_args}

Function: Translates (converts) a form definition segment into a machine readable form segment suitable for use with the vform_ subroutine entry points.

Arguments:

path
    is the pathname of the form definition segment to be converted. A ".form" suffix is required on the form definition segment. If a ".form" suffix is not specified on the pathname, it is assumed.

Control Arguments:

-brief, -bf
    causes error messages to contain a brief description of the error instead of the full error message text. The default is to output the long form of the error message the first time and the short form of the message each additional time that the same error message is used.

-debug, -db
    causes additional information about each field of the form to be printed on the screen. Be warned that there is considerable output from this control argument. It is designed to aid in translator debugging.

-default
    resets the values of all previously defined control arguments, that is, negates the effects of any other control arguments.

-long, -lg
    causes the text of the error messages produced to always contain the long form of the error message. The default is to output the long form of the error message the first time and the short form of the message each additional time that the same error message is used.

-list, -ls
    causes an expanded listing of each field defined in the form to be created. This listing contains each field and all statements for that field. A field that is defined "like" another field will not appear with a "like" statement, but will have each

property of the field explicitly defined. The listing segment name is the same name as the form input segment name, but with the ".form" suffix replaced with a ".list" suffix.

-meter, -mt
: causes some metering information about the searching for previously defined fields to be printed at the end of the translation. This information is primarily useful to developers of the forms software.

Notes

It should be noted that one error on the form definition input segment tends to propagate itself and cause other errors to appear later, especially if many fields are defined "like" other fields. Due to this "feature", one small error at the beginning of the input segment may cause literally hundreds of error messages.

Name: display_form, dform

Syntax:

display_form ‹path› {-control_args}

Function: Displays a form on the screen as it would appear if the form has been opened and displayed with vform_$display_form. The program also optionally clears the screen when it is done and optionally performs a vform_$read_form operation.

Arguments:

path
    is the pathname of a converted form segment. Archive component pathnames are supported.

Control Arguments:

-brief, -bf
    Causes the message "Processing of the form has been aborted." to be suppressed when -read is given and the user aborts the form.

-clear
    causes the program to clear the screen after the program is finished. This is the default if "-read" is specified.

-long, -lg
    Negates the operation of the -brief control argument. This is the default.

-no_clear
    causes the program not to clear the screen after the programs completion. This is the default if "-no_read" is specified.

-no_read
    causes the program not to perform the vform_$read_form operation after displaying the form. This is the default.

-output_file, -of
    causes the form to be displayed into a file in the working directory called ‹entry portion of path›.formout. Any blank unprotected fields will be displayed with "underscore" characters.

-read
    causes the program to perform a vform_$read_form operation after the form is displayed.

Notes:

This program is particularly useful when debugging a form. It allows the user to perform the vform_$display_form and/or vform_$read_form operations on the form without writing any software.

Name: list_forms, lforms

Syntax:

```
list_forms
```

Function: Lists all open forms. It also lists those forms that have been closed, but whose control segment entries have not been reused. This command takes no arguments.

Notes:

This command does not list all forms that have ever been opened. Rather, it lists all open forms, as well as those that have been closed, but their control segment entries not reused.

This command is useful in finding out whether or not an application has left any forms open. Forms should always be closed when their usefulness is over to free the considerable amount of storage occupied by an open form.

Name: print_form, pform

Syntax:

print_form path {-control_args}

Function: Given a converted form segment, the program produces a complete view of the form definition segment as it would look with all "like"s and expressions expanded and evaluated.

Arguments:

path
     is the pathname of a converted form segment. Archive component names are allowed.

Control Arguments:

-brief, -bf
     Suppresses the printing of the header and presents the data in a shorter form.

-header, -he
     Causes a header consisting of the information about the form (the form statements) to be printed along with the information about each field. This is the default if no specified fields are specified.

-header_only, -heo
     Causes only the header information to be printed.

-long, -lg
     Causes a header to be printed and the long information about each field to be printed.

-no_header_only, -noheo
     Negates the effect of the "-header_only" control argument.

-output_file path, -of path
     Causes the output to go into a file instead to the terminal. This control argument must be followed by an absolute or relative pathname of a segment to which to output should be directed.

Name: vform_create_include_file, vfcif

Syntax:

```
vform_create_include_file path {-control_args}
```

Function: Creates an include file containing two arrays and (optionally) a structure that can be used when referencing the names of fields in the form and their data. The two arrays are an array of names of fields for the form and an array to hold the data for these fields. If the structure is generated, it is a structure that overlays the data array and whose level 2 names correspond to the name array.

Arguments:

path
: is the pathname of a converted form segment. Archive component names are allowed.

Control Arguments:

-all, -a
: Causes all fields (protected and unprotected) to be included in the name array and the structure (if generated). The default is to include only unprotected fields.

-brief, -bf
: Inhibits the creation of a structure overlaying the data array with names corresponding to the name array.

-long, -lg
: Causes the creation of a structure overlaying the data array with names corresponding to the name array. This is the default.

-output_file path, -of path
: Causes the output file to be generated to have the specified name rather than the default name which is the name of form as specified by the "form" form statement. A ".incl.pl1" suffix is added if not present.

Section 6

Keyboard Functions

The forms package is designed to be compatible with a wide variety of terminals and to, whenever possible, make use of the terminals function keys. For this reason, it is not possible to give a complete list of all function keys and escape sequences for all terminals. The reason for this is that function keys generate escape sequences, and in order for a given terminal to use its function keys to their fullest potential, the terminal driver software sometimes has to redefine the meaning of a given escape sequence. What is provided is a list of the default key sequences to produce a given function. As a rule, these escape sequences will work on any terminal, but for the reason mentioned above, they may be different for certain terminals.

The notation used for control and escape sequences is the same as the notation used for the emacs text editor, but before this notation is defined, a description of control and escape sequences seems appropriate. A control sequence is a series of keystrokes that produces a control character. A control character is generated by depressing and holding the control key (sometimes marked CTL or CTRL on the keyboard) and then pressing another key. For example, to generate a control-A character, the user would depress and hold the control key and press the "A" key. Thus, a control character is generated much the same way as a "shifted" or "capital" letter.

An escape sequence is produced in a slightly different manner. An escape sequence is generated by pressing and releasing the escape key and then pressing and releasing another key. For example, to generate an escape-x, the user would depress and release the escape key (sometimes marked ESC) and then depress and release the "x" key. Thus, entering an escape sequence is like sending two distinct characters.

This distinction is crucial and thus should be committed to memory. A control sequence is like a "shifted" or "capital" letter, that is, depress and hold the control key while pressing another character. An escape sequence is similar to sending two characters, that is, depress and release the escape key then depress and release the other key.

The notation used for a control sequence is a caret or circumflex character (^) followed by the proper character. Thus, the notation ^X, pronounced "control X", would mean to depress and hold the control key while pressing the "X" key.

The notation used for an escape sequence is the letters "esc" followed by a dash ("-") followed by the additional character. Thus, the notation esc-B, pronounced "escape B", would mean to depress and release the escape key, then depress and release the "B" key.

Key sequences are composed of multiple escape and/or control sequences. For example, the key sequence for the "transmit_form" function is a ^X (control-X) followed by a ^C (control-C) whose notation would be "^X ^C" and which would be pronounced "control X control C". The key sequence for the "abort_form" function is the escape key followed by a control-Z whose notation is "esc-^Z" and is pronounced "escape control Z". Some terminal controllers, in order to support the function keys, will require even more complex escape sequences such as: "esc-- 1 ^Z ^V" which is pronounced "escape minus one control Z control V". Note that "esc--" means to depress and release the escape key then to depress and release the "-" key.

Available Functions and Keyboard Sequences

abort_form (esc-^Z): aborts processing of the form. The "normal" way to exit a form is with the "transmit_form" function (see below).

backtab (esc-A): moves the cursor to the first character position of the previous active, unprotected field of the form. If there is no previous active, unprotected field, then the terminal bell is rung and the function aborted.

backward_char (^B): moves the cursor backward one character position in the current field. If the cursor is already at the beginning of the current field, then one of several actions is performed. If the "hold_at_end" field attribute (See the "attribute" field statement in section 3) is enabled, then the terminal bell is rung and the function aborted. If the current field is a "next" field (See the "next" field statement in section 3) for another field, then the cursor is moved to the <u>last</u> character position of that field. Otherwise, the cursor is moved to the last character position of the previous active, unprotected field of the form if one exists. If not, the terminal bell is rung and the function is aborted.

backward_word (esc-B): moves the cursor backward one "word". A "word" is delimited by one or more characters that are neither numeric or alphabetic. If the current field is the "next" field of some other field, then the word may cross field boundaries.

```
           |---- Field Boundary ----|
field 1    Now is the time for all go
field 2    od men to come ...
```

If field 1 is connected to field 2, that is, field 1 has a "next" field statement of whose value is "field 2", then if the cursor was positioned after the "od" at the beginning field 2 (which is actually the word "good" broken between field 1 and field 2) and the user invoked the backward_word function, then the cursor would be positioned at the beginning of the "go" at the end of field 1.

beginning_of_field (^A): moves the cursor to the beginning of the current field.

delete_char (^D): deletes the character at the cursor and moves the characters in the field to the right of the cursor one space to the left. If the field has been defined in the form definition segment to have a "next" or connected field, then a character is moved from the beginning of the next field to the end of the current field. This process is repeated again if the next field has a "next" field until the list is exhausted.

end_of_field (^E): moves the cursor to the last position of the current field.

end_of_form (esc->): moves the cursor to the first character position of the last active, unprotected field in the form.

error_abort (^G): cancels the current key sequence. If the user begins a control sequence and realizes that an improper key sequence is being entered, then he may use the ^G key to cancel the processing of the sequence. The effect of this key is to ring the terminals bell and to abort the current function. This key may also be used to determine when the computer is ready to accept input from the terminal by hitting the ^G key and waiting for the "beep".

first_field_on_next_line (CR or RETURN): moves the cursor the the first character position of the first active, unprotected field on the next line (or row).

forward_char (^F): Moves the cursor one character position forward in the current field. If the cursor is already at the end of the current field, the action depends on several parameters. If the "hold_at_end" field attribute has been enabled, then the terminal bell will be rung and no further action taken. If the current field has a "next" field defined, then the cursor will be moved to the next active, unprotected field in the next list. Otherwise, the cursor is moved to first character position of the next active, unprotected field of the form, unless there are no more active, unprotected fields in the form in which case the terminal bell is rung and no further action is taken.

forward_word (esc-F): moves the cursor forward one "word". A word delimiter is any character that is neither alphabetic or numeric. See "backward_word" (Page 6-2) for a description of the behavior of word commands with respect to connected fields.

insert_mode_off (esc-I): turns off insert mode. With insert mode off, character typed when the cursor is over another character replace that character. Insert mode is off by default.

insert_mode_on (^X I): turns on insert mode. With insert mode on, characters typed when the cursor is over another character are inserted into the text before that character. If the current field has been defined to have a "next" field, then the character that was inserted off the end of the current field is inserted at the beginning of the "next" field. What happens to the last character of the chain of "next" fields is determined by the "overflow" field attribute. If the last field of the "next" field chain has the "^overflow" field attribute, then the terminal bell is rung, and the "sub_error_" condition is signalled with a specific error code. If there are no "next" fields, then the insertion is only within the current field. See the attribute field statement description in Section 3 for details. Insert mode is off by default.

kill_to_eof (^K): takes the data from the cursor position to the end of the current field and places it in a "kill ring" then deletes the data. The data placed on the "kill ring" may be retrieved using the "yank" function described below. The data may be "yanked" into the current field or into any other field.

next_line (^N): moves the cursor to the beginning of an active, unprotected field on the next line (row) of the form that is closest to the cursor position of the current field. If the next line (row) has no active, unprotected fields, then the next line (row) after that is used until a line (row) with an active, unprotected field is found. If there are no lines (rows) in the form past the current line (row) with active, unprotected fields, then the terminal bell is rung and the function aborted.

nop (^J): does nothing.

previous_line (^P): performs the same function as "next_line" above, but looks for active, unprotected fields on lines (rows) previous to the current line (row).

redisplay_form (^L): clears the screen and redisplays the current form with all of its active fields. This function is useful if data extraneous to the forms package has been inadvertently put on the screen, or if the screen has become garbled due to phone line noise, etc.

reset_form (esc-^L): resets the values and attributes of the form to the state in which the form was entered. This function is useful if the user has accidently destroyed some of the data of the form and wishes to restore it to the initial state.

rubout_char (DELETE (\177)): performs a "delete_char" function on the character to the left of the cursor.

tab (TAB (^I)): moves the cursor to the first character position of the next active, unprotected field of the form. If there are no more active, unprotected fields in the form, the terminal bell is rung and the function aborted.

top_of_form (esc-<): moves the cursor to the first active, unprotected field of the form.

transmit_form (^X ^C): "transmits" the data from the form to the applications program. It is possible for this function to not actually let the user exit the form if there is incorrect data in one of more of the fields. This action is caused by a "check_proc" or "exit_proc" finding invalid data in one or more fields. The proper action at this point is to correct the bad data and try to "transmit_form" again.

twiddle_chars (^T): exchanges the two characters to the left of the cursor. For example: "BA<cursor>" becomes "AB<cursor>" where <cursor> indicates the position of the cursor when the twiddle_chars functions is performed. There must be at least two characters to the left of the cursor in the current field or the terminal bell will be rung and the function aborted.

yank (^Y): Retrieves data that has been placed in the "kill ring" with the "kill_to_eof" function described above.

Appendix A

Form Definition Design and Efficiency

There are several techniques which can be used to increase the speed and efficiency of form definition. These techniques are not necessarily designed to increase the program efficiency when converting a form, but to help the designer to produce the form more efficiently. Several of these techniques will be discussed below.

Preplanning

One of the most effective ways of increasing the speed of form development is to first develop the form on paper. A standard coding form or other paper with a some type of grid works nicely.

Screen Layout

When laying out the form, there are several things to keep in mind. First, whenever possible, group related fields in the same area of the screen. The layout of the fields on the form should be so that the fields seem logical to the user, not in the easiest way for the program to interpret them. That is, order the fields so that the user may enter the data in the way the user's data is organized, not in a manner that is easy for the program to understand. Remember that forms applications are often targeted for users with little or no computer knowledge or experience.

Blocks of Fields

Once the form is laid out, the designer should then look over the form and break the fields into blocks, grouping similar fields and blocks of fields together. Then, when the fields are defined, they can be defined relative to the corner of these blocks using the "like" capability. Defining fields in this manner allows entire groups of fields to be moved around on the form simply by changing the location of the corner or origin of the block.

Attributes

Some forms designers feel that it is prudent to define "dummy" fields, that is, fields that are not active (^active), for the common types of fields such as data_fields, title_fields, error_fields, etc. Then, when a field is defined, its attributes are defined "like" the field of appropriate type. This gives the ability to change the attributes of all data or title or error fields simply by changing one statement. Thus the designer can experiment with different screen or form attributes for the various types of fields with little change to the form definition segment.

Errors

When defining a form, especially when defining a form with many "like" statements, it is likely that many error messages will be produced when the form is converted to binary with the cv_form command. This is due to the fact that if a field has a slight error and has fields defined "like" it and those fields have other fields "like" them, then each of the fields will get one or more error messages. So don't be alarmed if your first attempt at converting a form produces literally hundreds of error messages. When going through the error messages, it is often useful to correct the first few errors and then attempt to re-convert the form.

Appendix B

Program Efficiency Using Forms

```
+---------------------+
| ‹‹To Be Supplied›› |
+---------------------+
```

Appendix C

Writing a Terminal Controller

Support of terminals in Vform is accomplished via terminal-dependent subroutines. Vform attempts to locate by using the regular search rules, based on the terminal type maintained by Multics.

To support a type of terminal not supported by a supplied terminal controller, a new terminal controller must be written. A terminal controller is written as a PL/I source program, named vf_TTYTYPE_ctl_.pl1. If this terminal type is in you site's Terminal Type File (TTF), the name chosen should appear the same as it appears in the TTF, except that the name of the terminal controller should be all lowercase.

Terminal controllers are usually written by example from supplied terminal controllers. Once the terminal controller is written, it must be compiled before it can be used. Compilation is performed via the PL/I compiler, pl1. A typical command line to compile a terminal controller is:

```
     pl1 vf_vt100_ctl_
```

This produces an object segment, vf_vt100_ctl_

The most effective method of writing a new terminal controller is to take one that was written for a similar terminal and modify it. Almost all of the extant terminal controllers where written in this way. The sources are PL/I source segments, generally 10-20 printed pages long. Good starting points are:

vf_vt100_ctl_, typical of terminals that do not have the ability to insert or delete lines or characters.

vf_nsa7000e_ctl_, typical of terminal that do have insert or delete lines or characters. The two facilities are independent, either one, both, or neither may be present.

vf_tek4023_ctl_, typical of terminals that require space on the screen for display attributes. More about this is discussed in the section entitled "DISPLAY ATTRIBUTES HANDLING".

ENTRY POINTS

The entry points in the terminal controller are standardized. They have the same names in all terminal controllers. The Vform screen manager calls these subroutines anonymously after the appropiate terminal controller has been initialized.

REQUIRED ENTRY POINTS

Entry point init

entry point dcl: init: entry ();

The init entry point is called a form open time; it has the responsibility of setting various flags, and initializing the terminal. The init entry point has the responsibility for initializing the following structure. It is declared in vf_tty_info.incl.pl1.

```
dcl  1 tty_info                        aligned based (vfs_$tty_info_ptr
       2 version                       fixed bin(35),
       2 terminal_type                 char(32) unaligned,
       2 size                          aligned,
         3 height                      fixed bin(17) unaligned,
         3 width                       fixed bin(17) unaligned,
       2 flags                         aligned,
         3 insert_charsp               bit(1) unaligned,
         3 delete_charsp               bit(1) unaligned,
         3 ctl_will_manage_modesp      bit(1) unaligned,
         3 use_display_fieldp          bit(1) unaligned,
         3 interpret_stringp           bit(1) unaligned,
         3 return_attribute_stringsp   bit(1) unaligned,
         3 wipe_attributes_with_spacesp bit(1) unaligned,
         3 protect_is_display_attrp    bit(1) unaligned,
         3 mbz                         bit(29) unaligned,
       2 position                      aligned,
         3 row                         fixed bin unaligned,
         3 column                      fixed bin unaligned,
       2 bell                          entry (),
       2 clear_attributes              entry (),
       2 clear_screen                  entry (),
       2 clear_to_end_of_line          entry (),
       2 clear_to_end_of_screen        entry (),
       2 delete_chars                  entry (fixed bin),
       2 display_field                 entry (ptr, ptr),
       2 display_text                  entry (char(*)),
       2 get_one_unechoed_char         entry () returns (char(1)),
       2 insert_text                   entry (char(*)),
       2 interpret_string              entry (char(*) varying, fixed bi
       2 position_cursor               entry (fixed bin unal, fixed bin
       2 return_clear_attributes_string entry (char(*) varying),
       2 return_set_attributes_string  entry (ptr, char(*) varying),
       2 set_attributes                entry (ptr),
        2 set_forms_modes                entry (),
        2 unset_forms_modes              entry ();
```

version
    the version will be set by the Vform screen manager and should be checked for TTY_INFO_VERSION_1.

terminal_type
    should be set to the terminal type that his controller supports.

height
    should be set to the maximum number of lines that the terminal has on its display.

width
    should be set to the maximum number of characters across one line.

insert_charsp
    if set, indicates that character insertion is permitted/implemented.

delete_charsp
    if set, indicates that character deletion is permitted/implemented.

ctl_will_manage_modesp
    if set, indicates that there are entry points that can be called to set and reset the Multics terminal modes.

use_display_fieldp
    if set, indicates that the Vform screen manager should call display_field instead of the set_attributes, display_text, clear_attributes combination.

interpret_stringp
    if set, indicates that there is a entry point that can be called to interpret input from the terminal (function key implementation.)

return_attribute_stringsp
    if set, indicates that there are entry points that can be called to obtain the proper character strings to set the desired display attributes.

wipe_attributes_with_spacesp
    if set, indicates that the terminal when displaying over a position on the screen will change the display attributes, of that position, to the current display attributes in effect.

protect_is_display_attrp
    if set, indicates that the controller wants to use protect as a display attribute. This should only be used in the case where the terminal does not have multiple display attributes. An example of

this is the VIP7201 controller, and because it only has one display attribute only non-protected fields are highlighted.

mbz
    must be set to "0"b.

row
    is the current row position of the cursor. (Does not need to be set at initialization).

column
    is the current column position of the cursor. (Does not need to be set at initialization).

bell
    should be set to the entry point to be called when the terminals bell is to be rung. (See hammer.)

clear_attributes
    should be set to the entry point to be called when the screen attributes are to be set to their default state.

clear_screen
    should be set to the entry point to be called when the screen is to be cleared.

clear_to_end_of_line
    should be set to the entry point to be called when the screen is to be cleared from the current screen position to the end of the current line.

clear_to_end_of_screen
    should be set to the entry point to be called when the screen is to be cleared from the current screen position to the end of the screen.

delete_chars
    should be set to the entry point to be called when characters are to be deleted from the screen. Notes: this only needs to be set if tty_info.flags.delete_charsp has been set.

display_field
    should be set to the entry point to be called when a field is to be placed on the screen. Note: this only needs to be set if tty_info.flags.use_display_fieldp has been set.

display_text
    should be set to the entry point to be called when text is to be placed on the screen.

get_one_unechoed_char
    should be set to the entry point to be called when one character is to be input from the terminal.

insert_text
should be set to the entry point to be called when text is to be inserted into the screen. Note: This only needs to be set if tty_info.flags.insert_charsp has been set.

interpret_string
should be set to the entry point to be called when input from the terminal is to be interpreted. Note: This only needs to be set if tty_info.flags.interpret_stringp has been set.

position_cursor
should be set to the entry point to be called when cursor positioning.

return_clear_attributes_string
should be set to the entry point that will return a character string that will clear the display attributes. Note: this only needs to be set if tty_info.flags.return_attribute_stringp has been set.

return_set_attributes_string
should be set to the entry point that will return a character string that will set the display attributes. Note: this only needs to be set if tty_info.flags.return_attribute_stringp has been set.

set_attributes
should be set to the entry point that will be called to set the current display attributes.

set_forms_modes
should be set to the entry point that will be called to set the Multics terminal modes needed by the Vform package. Note: this only needs to be set if tty_info.flags.ctl_will_manage_modesp is set.

unset_forms_modes
should be set to the entry point that will be called to reset the Multics terminal modes that where needed by the Vform package. Note: this only needs to be set if tty_info.flags.ctl_will_manage_modesp is set.

Entry point bell

entry point dcl: bell: entry ();

The bell entry point rings the terminal's bell. In most cases this only requires sending 007o to the terminal.

Entry point clear_attributes

entry point dcl: clear_attributes: entry ();

The clear_attributes entry point sends the character string required by the terminal to clear all the terminal's display attributes. More about attribute handling can be found in the section entitled Attribute Handling.

Entry point clear_screen

entry point dcl: clear_screen: entry ();

The clear_screen entry point sends the character string required by the terminal to clear the terminal's string. It is assumed by the Vform screen attributes manager that at the same time the terminals display attributes are cleared also. If the terminal does not have an explicit clear_screen escape sequence then you must explicitly clear the screen by hand.

Entry point clear_to_end_of_line

entry point dcl: clear_to_end_of_line: entry ();

The clear_to_end_of_line entry point sends the character string required by the terminal to clear the line from the current cursor position to the end of the current line. If the terminal does not have an explicit clear_to_end_of_line escape sequence then you must explicitly clear from the current cursor position to the end of the current line.

Entry point clear_to_end_of_screen

entry point dcl: clear_to_end_of_screen: entry ();

The clear_to_end_of_screen entry point sends the characters string required by the terminal to clear the screen from the current cursor position to the end of the screen. If the terminal does not have an explicit clear_to_end_of_screen escape sequence then you must explicitly clear from the current cursor position to the end of the screen.

Entry point display_text

entry point dcl: display_text: entry (A_text);

The display_text entry point places the specified text on the screen.

A_text
The characters to be placed at the current screen location. Input (char (*)).

Note that this entry point does not worry about terminal display attributes.

Entry point get_one_unechoed_char

entry point dcl: get_one_unechoed_char: entry () returns (char (1));

The get_one_unechoed_char entry point should get and return one character of input.

Entry point position_cursor

entry point dcl: position_cursor: entry (A_new_row, A_new_column);

The position_cursor entry point should send the character string neccessary to position the terminal's cursor to the location specified.

A_new_row
Will be the row the cursor is to be placed on. (Input (fixed binary unaligned)).

A_new_column Will be the column the cursor is to be placed on. (Input (fixed binary unaligned)).

Entry point set_attributes

entry point dcl: set_attributes: entry (A_field_ptr);

The set_attributes entry point should send the character string neccessary to enable the terminal's display attributes requested.

A_field_ptr
is a pointer to the information for the field that the terminal attributes are to be set from. The important section of this structure is described in the section "DISPLAY ATTRIBUTES HANDLING". (Input (pointer)).

OPTIONAL ENTRY POINTS

These optional entry points are supplied to allow the Vform screen manager to use the special features of a terminal and to be able to manage different types of terminal. All of these entry points are controlled by flags in the tty_info structure. If the corresponding flag is set then the entry point must exist.

Entry point delete_chars

entry point dcl: delete_chars: entry (A_nchars);

The delete_chars entry point should send the terminal's delete_char control sequence to the terminal.

A_nchars
is the number of characters to be deleted (Input (fixed binary)).

This entry point will only be called by the Vform screen manager when tty_info.flags.delete_charsp has been set to "1"b.

Entry point display_field

entry point dcl: display_field: entry (A_field_ptr, A_old_field_ptr);

The display_field entry point should display an entire field on the screen. This includes the display attributes for the field, the field text, and the display attributes needed to clear the display attributes.

A_field_ptr
is a pointer to the field to be displayed on the screen. The structure pointed to by this pointer is declared in vf_field.incl.pl1. (Input (pointer)).

A_old_field_ptr is a pointer to the old field information of the field to be displayed on the screen. The structure pointed to by this pointer is declared in vf_field.incl.pl1. (Input (pointer)).

This entry point is only called by the Vform screen manager if the tty_info.flags.use_display_fieldp has been set to "1"b. This entry point can also be used in the case that the terminal display attributes require a character location of the screen. For more information see the section entitled Attribute Handling.

Entry point insert_text

entry point dcl: insert_text: entry (A_text);

The insert_chars entry point should send the terminal's insert_on control sequence to the terminal, the text to be inserted, and then the insert_off control sequence.

A_text
is the text to be inserted. (Input (char (*)).

The insert_text entry point will only be called by the Vform screen manager when tty_info.flags.insert_charsp has been set to "1"b.

Entry point interpret_string

entry point dcl: interpret_string (A_char_string, A_action_code);

The interpret_string entry point interprets input sent by the user from the terminal. This entry point is used primarily to bind the terminals function keys to Vform actions.

A_char_string
is the character string to be interpreted. (Input (char (*) varying)).

A_action_code
should be set to the action that the Vform input manager is to perform.

This entry point is called with a character string and should process the character string and return an action code. The following actions codes are declared in vf_action_codes.incl.pl1.

vfs_$more
indicates that the character string is a part of a known control sequence, but more characters are needed to determine the exact action to be performed.

vfs_$no_action
indicates that the character string matches no known control sequence and should be handled by the Vform input controller or discarded.

vfs_$nop
indicates that the character string is valid but nothing should be done.

vfs_$error_abort
indicates that the previous input (being processed) was in error and should be discarded (Vform input manager will ring the terminal's bell.)

vfs_$backtab
indicates that the character string maps into a backtab operation.

vfs_$backward_char
indicates that the character string maps into a move cursor left operation.

vfs_$beginning_of_field
indicates that the character string maps into a go to beginning of field operation.

vfs_$delete_char
indicates that the character string maps into a delete character operation.

vfs_$end_of_form
indicates that the character string maps into a go to end of form operation.

vfs_$forward_char
indicates that the character string maps into a move cursor right on character operation.

vfs_$next_line
indicates that the character string maps into a move cursor to next line operation.

vfs_$previous_line

indicates that the character string maps into a move cursor to previous line operation.

vfs_$tab
indicates that the character string maps into a tab or next field operation.

vfs_$top_of_form
indicates that the character string maps into a go to top of form operation.

vfs_$end_of_field
indicates that the character string maps into a go to the end of this field operation.

vfs_$first_field_on_line
indicates that the character string maps into a go to the first field on this line operation.

vfs_$twiddle_chars
indicates that the character string maps into a twiddle operation.

vfs_$kill_to_eof
indicates that the character string maps into a kill to end of field operation.

vfs_$rubout_char
indicates that the character string maps into a delete character operation.

vfs_$first_on_next_line
indicates that the character string maps into a go to first field on the next line operation.

vfs_$forward_word
indicates that the character string maps into a move cursor to begining of next word operation.

vfs_$backward_word
indicates that the character string maps into a move cursor to the begining of the previous word operation.

vfs_$insert_mode_on
indicates that the character string maps into a request that insert mode be enabled.

vfs_$insert_mode_off
indicates that the character string maps into a request that insert mode be disabled.

vfs_$redisplay_form
indicates that the character string maps into a request that the form be redisplayed.

vfs_$reset_form
indicates that the character string maps into a request that the form be reset.

vfs_$transmit_form
indicates that the character string maps into a request that the form be considered transmitted. (Actually control returned the user application.)

vfs_$yank
indicates that the character string maps into a request to "yank" previously deleted text.

vfs_$abort_form
indicates that the character string maps into an abort form request.

Care should be that vfs_$more is only passed back when the current character string is a subpart of a valid control sequence. Remember, this entry point gets called whenever a character is typed by the user. This entry point is only called when tty_info.flags.interpret_stringp is set to "1"b.

<u>Entry point</u> return_clear_attributes_string

entry point dcl: return_clear_attributes_string: entry (A_char_string);

The return_clear_attributes_string entry point should return the character string which will clear all display attributes.

A_char_string
    should be set to the character string that when sent to the terminal will clear all the terminal's display attributes. (Output (char (*) varying)).

This entry point is only call if tty_info.flags.return_attributes_string is set to "1"b.

<u>Entry point</u> return_set_attributes_string

```
entry point dcl:  return_set_attributes_string:  entry (A_field_ptr,
                                                         A_char_string);
```

The return_set_attributes_string entry point should return the character string which will set all the desired display attributes.

A_field_ptr
    is a pointer to the information for the field that the terminal attributes string is to be set from. The important section of this structure is described in the section entitled "DISPLAY ATTRIBUTES HANDLING". (Input (pointer)).

A_char_string
    should be set to the character string that, when sent to the terminal will set the required terminal display attributes.

This entry point is only called if tty_info.flags.return_attributes_string is set to "1"b.

Entry point set_forms_modes

entry point dcl: set_forms_modes: entry ();

The set_forms_modes entry point should set the tty_ modes for communication with the terminal that the Vform package requires. Vform requires the following tty_ modes:

```
init,breakall,rawo,rawi,ctl_char,force,fulldpx,
   hndlquit,^ll,^pl,^oflow
```

This entry point should save the current tty_ modes for restoration by unset_forms_modes. This entry point will only be called if tty_info.flags.ctl_will_manage_modesp is set to "1"b. If tty_info.flags.ctl_will_manage_modesp is set to "0"b, the Vform screen manager will attempt to set the proper tty_ modes.

Entry point unset_forms_modes

entry point dcl: unset_forms_modes: entry ();

The unset_forms_modes entry point should set the tty_ modes for communication with the terminal to what was saved in the set_forms_modes entry point. This entry point will only be called if tty_info.flags.ctl_will_manage_modesp is set to "1"b. If tty_info.flags.ctl_will_manage_modesp is set to "0"b, then the Vform screen manager will attempt to restore the proper tty_ modes.

Entry point video_system_init

entry point dcl: video_system_init: entry ();

The video_system_init entry point should initialize the tty_info structure like the init point does. However, the Vform screen manager understands how to use the video system for terminal dependent functions, so most of the entry points mentioned above will not be used. If you are planning to use the video system with this controller then the return_set_attributes_string and return_clear_attributes_string should exist (and work) for efficiency, but they are not needed. However, if return_set_attributes_string and return_clear_attributes do not exist then the set_attributes and the clear_attributes entry points should exist. The video_system_init entry point will only be called if the video system is invoked.

inverse

indicates that the characters should be display in inverse (black letters on bright background, sometimes called inverse video.) Most terminals support this display attribute.

dotted_underlined

indicates that the characters should be displayed with a dotted underline underneath them. If the terminal does not support this display attribute it should be mapped to an underline attribute (if the underline attribute does not exist it should be mapped to a inverse attribute.)

blink

indicates that the characters when displayed should blink. If the terminal does not support this display attribute it should be mapped to an inverse attribute.

Something to consider also, is that display attributes can be combined. Some terminals allow this and some don't. You will just have to play with the mappings and combinations until things work the way you want them to.

Appendix D

vform_ Subroutine Entry Point Include Files

Each vform_ entry point has a corresponding include file that defines an internal subroutine that can be used instead of the call to vform_. This method has several advantages. The code produced by the compiler is smaller if multiple calls to the same entry point are present because the compiler can generate internal calls to the subroutine and only one full blown external call to the vform_ entry point. When multiple calls to the same routines are used, the compiled code is smaller and compile time is reduced.

Another advantage of using the internal subroutines contained in the include files is that the caller does not have to worry about checking the error codes. If the error code was non-zero, then the system subroutine sub_err_ is called with appropriate arguments to halt execution of the program and place the user at a new command level.

The following is a list of the names of the subroutines that are defined, the corresponding vform_ entry point, the include file which contains the subroutine, and any notes about the action or parameters to the subroutine. Unless otherwise specified, the parameters to the subroutines are exactly the same as the corresponding vform_ subroutine entry point described in Section 4 but the "code" argument is omitted.

Subroutine: assign_values
vform_ Entry Point: vform_$assign_values
Include File: vform_assign_values.incl.pl1
Notes: Both the 'error_index' and 'code' parameters must be omitted.

Subroutine: class_blank_field_count
vform_ Entry Point: vform_$class_blank_field_count
Include File: vform_class_blank_count.incl.pl1

Subroutine: class_is_all_blank
vform_ Entry Point: vform_$class_is_all_blank
Include File: vform_class_blank.incl.pl1

Subroutine: class_is_all_non_blank
vform_ Entry Point: vform_$class_is_all_non_blank
Include File: vform_class_non_blank.incl.pl1

Subroutine: class_non_blank_field_count
vform_ Entry Point: vform_$class_non_blank_field_count
Include File: vform_class_nonblnk_cnt.incl.pl1

Subroutine: get_value
vform_ Entry Point: vform_$get_value
Include File: vform_get_value.incl.pl1

Subroutine: open_form
vform_ Entry Point: vform_$open_form
Include File: vform_open_form.incl.pl1

Subroutine: position_cursor
vform_ Entry Point: vform_$position_cursor
Include File: vform_position_cursor.incl.pl1

Subroutine: read_form
vform_ Entry Point: vform_$read_form
Include File: vform_read_form.incl.pl1
Notes: The 'code' parameter must not be omitted from this routine, but its value will only be returned as 0 or vf_et_$form_aborted. Any other error code will cause it to abort.

Subroutine: read_transmit_form
vform_ Entry Point: vform_$read_transmit_form
Include File: vform_read_transmit.incl.pl1
Notes: The 'code' parameter must not be omitted from this routine, but its value will only be returned as 0 or vf_et_$form_aborted. Any other error code will cause it to abort.

Subroutine: reset_class_modified_flag
vform_ Entry Point: vform_$reset_class_modified_flag
Include File: vform_reset_class_flag.incl.pl1

Subroutine: reset_form
vform_ Entry Point: vform_$reset_form
Include File: vform_reset_form.incl.pl1

Subroutine: reset_modified_flag
vform_ Entry Point: vform_$reset_modified_flag
Include File: vform_reset_flag.incl.pl1

Subroutine: set_attributes
vform_ Entry Point: vform_$set_attributes
Include File: vform_set_attributes.incl.pl1

Subroutine: set_attributes_and_value
vform_ Entry Point: vform_$set_attributes_and_value
Include File: vform_set_attr_value.incl.pl1
Notes: The set_attributes_and_value combines the function of the vform_$set_attributes and the vform_$set_value subroutines in one call. The first argument is the form_index, the second one is the field name, the third argument as an ‹attributes_string› and the fourth argument is the new value for the field. The fourth argument may be an ioa_ control

string, in which case there may be additional argument to accommodate it. See the ioa_ subroutine in the manual 'Multics Subroutines and I/O Modules' for details.

Subroutine:	set_class_attributes
vform_ Entry Point:	vform_$set_class_attributes
Include File:	vform_set_class_attr.incl.pl1
Subroutine:	set_class_value
vform_ Entry Point:	vform_$set_class_value
Include File:	vform_set_class_value.incl.pl1
Subroutine:	set_value
vform_ Entry Point:	vform_$set_value
Include File:	vform_set_value.incl.pl1
Subroutine:	test_attributes
vform_ Entry Point:	vform_$test_attributes
Include File:	vform_test_attributes.incl.pl1
Subroutine:	test_class_attributes
vform_ Entry Point:	vform_$test_class_attributes
Include File:	vform_test_class_attributes.incl.pl1
Subroutine:	update_form
vform_ Entry Point:	vform_$update_form
Include File:	vform_update_form.incl.pl1

To use these subroutines, simply include the appropriate include files using the "%include" pl1 statement and then call the internal subroutine that is defined. For example, to use the open_form subroutine, the following code could be used:

```
%include vform_open_form;

call open_form (form_path, form_index);
```

and to use the assign_values subroutine:

```
%include vform_assign_values;

call assign_values (form_index, name_array, data_array);
```

Note that the "%include" statement need only be inserted once per subroutine. Programming convention is to place all "%include" statements at the very end of the program, or at the end of the list of declares.