MULTICS TECHNICAL BULLETIN MTB- 405

To: MTB Distribution

From: Gary C. Dixon

Date: January 16, 1979

Subject: New query_ Subroutine, and a
Proposed Set of Argument Validation Active Functions

Introduction

A central feature of the new Trouble Report System are the commands, enter_trouble_report, add_to_trouble_report, and answer_trouble_report. Like trouble_report (their predecessor), these commands prompt the user for information needed to enter a report, add to an existing report or answer a report. Unlike trouble_report, these commands must have the ability to parse up a pre-typed input segment to obtain their information, rather than asking questions. Also, they must allow the user to edit his input before actual submission (to correct typos, add forgotten details, etc). A new query_ subroutine has been developed to provide a centralized set of interfaces for asking questions, storing the answers in a segment which can be edited, and reparsing the segment to obtain corrected answers.

Proposed Solution

The query_ subroutine is described in detail by the MPM documentation which follows. It provides a mechanism for defining questions, grouping them into units in which all questions are asked by a single call to query_, or are answered by parsing a single input segment. Formatted answers can then be placed in an output segment for subsequent processing.

Because query_ is attempting to perform a rather complicated job, its interface is not as simple as that of ask_ or command_query_. However, as the sample program at the end of the query_ documentation illustrates, query_ is not difficult to use. Your comments on the function being performed, the interfaces suggested, or other enhancements to query_ will be appreciated. query_ exists now for experimental use on System M. If necessary, it can be carried to MIT.

Miscellaneous

In the design of answer validation routines for query_, it became apparent that we are missing an important set of active functions, those which validate the format and value of various kinds of objects. Following the MPM description of query_ is an info segment describing five possible active functions: valid_date, valid_number, valid_pathname, valid_picture, and valid_word. These have not been implemented as yet. Your comments on their proposed interface, or upon other possible validation active functions will be appreciated.

In writing query_, I was unable to find reasonable error_table_ codes to describe several types of data manipulation failure. These codes are listed as query_et_ values in the query_ documentation, but will probably be added to error_table_ should query_ be installed. These query_et_ codes are described following the validation active function info segments.

Name: query_

The query_ subroutine is a generalized question asking facility. The subroutine asks questions of the user, validates the answers and returns the answers to its caller. query_ can also parse an input segment looking for the answers to questions.

query_ is designed to ask and answer many questions at a time. The caller provides information about each question to be asked, including: a long and a short version of the question; an information string describing the intent of the question and possible answers; an array of delimiters, any one of which can be used to separate the question from its answer; an array of delimiters, any one of which can be used to end the answer; an array of answer processing routines which can redefine the given answer; and a validation routine which verifies the correctness of the answer.

query_ stores information about each possible question in a query data base created in the process directory. Once the questions are defined, the caller can group questions together into units. A single call to query_ asks all of the questions in a unit, and returns all of the answers. Based upon those answers, questions in other units can be asked until all appropriate information is obtained.

Besides asking questions, query_ can look in a formatted input segment for answers to questions in a unit. Also, answers found by asking questions or parsing an input segment can be stored in a formatted output segment suitable for dprinting, mailing, etc. At a later point in time, query_ can parse the formatted segment into separate answers for subsequent processing.

The query_ subroutine has several entry points which perform its various functions. query_$init must be called first to initialize the query data base in the process directory. Then query_ must be called to define each question to be asked. query_$ask can then be called to ask the user individual questions. query_$add_unit can be called to group a series of questions into a unit. Then query_$parse_unit can be called to parse an input segment into the answers for one or more question units. Or query_$ask_unit can be called to ask the questions in one or more units. query_$format_unit can be called to store the answers for one or more question units in a formatted output segment. Finally, query_$term must be called to terminate use of the query data base. These entry points and several others are described below. An example is shown following the entry point descriptions.

Entry: query_$init

This entry point is called to initialize the query data base in the process directory. Each caller of query_ must use a separate data base. The data base can be used in a single process for as long as necessary, but should be terminated when no longer needed by calling query_$term.

Usage

```
     declare query_$init entry (char(*), ptr, fixed bin(35));

     call query_$init (caller, Pq, code);
```

where:

1. caller (Input)
   is the name of the program on whose behalf the questions are being asked.

2. Pq (Output)
   points to the query data base.

3. code (Output)
   is a standard status code. It may have any value returned by the define_area_ subroutine.

Entry: query_

This entry point is called to define one of the possible questions which may be asked. Each question must be defined before it can be asked or grouped into a unit, or before its answer can be searched for in an input segment. However, the query_ entry point can be called at any time to define a new question.

When a question is defined, the user assigns an identifier by which it can be referenced in subsequent calls to query_ entry points. The identifiers are positive integers which must be assigned in increasing numeric order, beginning with 1. No integer identifiers may be skipped.

Associated with each question are a long and short version of the question. The user is asked the long version of the question unless a brief switch is on in the call to query_$ask or query_$ask_unit. Either the long or short version of the question may be used as a title to identify an answer found in a formatted input segment when query_$parse_unit is called.

Also associated with the question is an information string. The information string describes the intent of the question, and gives possible answers to prompt the user for the correct input. When query_$ask or query_$ask_unit ask a question which the user does not understand or does not know how to answer, he can type an information prompt string (e.g., a line containing just a ?) which causes the information string to be printed. The question is then reasked.

An array of question delimiters specifies what delimiters can be used to separate a question from its answer in an input segment. Also, the first delimiter in the array is used to end the question when the user is asked a question. Note that when the user is asked a question, query_$ask and query_$ask_unit do not add a newline character after the question delimiter. In this way a question can appear on the same line as the answer typed by the user. Questions for which long answers are expected may have a delimiter ending in a newline so that all lines of the answer can be aligned under the question.

An array of answer delimiters specifies what delimiters can be used to end an answer. Several delimiters are allowed so that a variety of ending conventions can be accommodated. For example, a multiline answer could be delimited by a line containing just a period (<NL>.<NL>), or by two blank lines (<NL><NL><NL>). A one word answer could be delimited by a newline, space or horizontal-tab. This would permit several questions with one word answers to appear on the same line in a formatted input file. Also, query_$ask and query_$ask_unit can handle answer type-ahead. By using space or horizontal-tab delimiters, several one word answers can be given on the same line when the user knows the questions in advance.

An array of answer processing subroutines can be provided to pre-process the answer before it is validated. Typical pre-processing might include expansion of abbreviations or symbols in the answer, expansion of active functions and iteration parentheses, answer formatting, etc. The pre-processing feature is currently not implemented.

Finally, a validation routine is provided to verify that the proper answer is given for the question. query_ supplies several validation routines, as described under "Validation Routines" below. The caller can provide other routines to validate specially-formatted answers.

Usage

```
     declare query_ entry (ptr, fixed bin, char(*), char(*), char(*),
          (*) char(*) varying, (*) char(*) varying, (*) entry,
          (*,*) char(*) varying, entry, (*) char(*) varying, fixed bin(35));

     call query_ (Pq, qid, qlong, qshort, qinfo, qdelims, adelims, aprocessors,
          aproc_args, avalidator, avalid_args, code);
```

where:

1. Pq (Input)
   points to the query data base.

2. qid (Input)
   is the question identifier. It must be a positive integer. The first question must have an identifier of 1, the second of 2, and so on.

3. qlong (Input)
   is the long version of the question.

4. qshort (Input)
   is the short version of the question.

5. qinfo (Input)
   is the information string describing the intent of the question, and listing possible answers.

6. qdelims (Input)
   is an array of question delimiters. The first delimiter is used by query_$ask and query_$ask_unit to end the question when asking the user a question. Other delimiters are available as optional question delimiters in the input segments parsed by query_$parse_unit.

7. adelims (Input)
is an array of answer delimiters. The user can use any of these delimiters to end the answer to his question.

8. aprocessors (Input)
is an array of answer pre-processing subroutines. Answer pre-processing is not currently implemented. The nothing subroutine can be used as an argument holder for this argument.

9. aproc_args (Input)
is a 2-dimensional array of character string arguments which are passed as data to the answer pre-processing routines. The array should be dimensioned as follows:

```
          dcl aproc_args (no_aprocessors, no_args_to_aprocessor)
                    char(*) varying;
```

Answer pre-processing is not currently implemented. A null character string ("") can be passed as an argument holder for this argument.

10. avalidator (Input)
is an entry variable identifying a validation routine which can verify the correctness of the answer. See "Validation Routines" below for further details.

11. avalid_args (Input)
is an array of character string arguments which are passed as data to the validation routine. See "Validation Routines" below for further details.

12. code (Output)
is a standard status code which indicates the failure of question definition. The following value may be returned.

error_table_$key_order
the question identifier (qid) given in this call is not one larger than the last id which was used. Identifiers must be positive integers, beginning with one and used in sequential order.

Validation Routines

Each time a question is asked by query_$ask or query_$ask_unit, or is found by query_$parse_unit, its answer is validated for correctness by calling a validation routine. A validation routine is a subroutine which has the calling sequence shown below.

```
     declare avalidator entry (ptr, fixed bin, ptr, char(*),
          (*) char(*) varying, fixed bin(35)) returns (bit(1));

     true_false = avalidator (Pq, qid, Pvalid_ctl, answer, avalid_args, code);
```

where:

1. Pq (Input)
points to the query data base.

2. qid (Input)
   is the identifier of the question being validated.

3. Pvalid_ctl (Input)
   points to the valid_ctl structure described below. This structure contains information used by the validation routine.

4. answer (Input)
   is the answer to be validated. The text of this answer may not be changed in any way.

5. avalid_args (Input)
   is an array of character string arguments which the caller of query_ passed as data to the validation routine. This data may control the operation of some validation routines, or may list specifications for valid answers, etc. Each validation routine may interpret these arguments in its own way.

6. code (Output)
   is a standard status code describing the failure of the validation routine. If a nonzero value is returned, then all questioning stops and the code value is returned to the caller of query_$ask, query_$ask_unit or query_$parse_unit. If a zero value is returned, then the validity of the answer is indicated by the true_false return value.

7. true_false (Output)
   when = "1"b, indicates that the answer is valid. When = "0"b, the answer is invalid. A value of "0"b should be returned when a nonzero code is also returned.

The Pvalid_ctl argument of the validation routine points to the structure shown below. This structure is declared in query_valid_ctl_.incl.pl1.

```
dcl  1 valid_ctl                    aligned based(Pvalid_ctl),
       2 version                    fixed bin,
       2 pad1                       bit(36),
       2 Perror_iocb                ptr,
     Pvalid_ctl                     ptr,
     Vvalid_ctl_1                   fixed bin int static
                                         options(constant) init (1);
```

where:

1. version
   is the version number of this structure. It is currently 1. See the description of Vvalid_ctl_1 below.

2. pad1
   is reserved for future use. The caller must set this to ""b.

3. Perror_iocb
   points to the I/O Switch Control Block (IOCB) through which an error can be reported to the user. The IOCB must be opened for stream_output.

4. Pvalid_ctl
   points to the valid_ctl structure.

5. Vvalid_ctl_1
   is a named constant which should be used to check for a structure version number of 1.

The query_ facility provides four validation routines which are described below. In addition, the caller of query_ may provide routines to perform specialized types of validation.

1. query_$no_validation
   performs no validation whatsoever. Any answer is valid, including a null string.

2. query_$any_value
   requires that some (nonnull string) value be given as the answer. No further validation is performed for the value.

3. query_$list_validation
   requires that the answer be a single word which appears in the list of acceptable words passed in the avalid_args array. The array may have one or more elements, each of which is a list of acceptable words. Each word, including the first, must be preceded and followed by a space character. For example, the list

   " yes no maybe "

   defines three acceptable words: "yes", "no" and "maybe". If more than one element is given in the avalid_args array, the elements are logically combined into a single, large list. The leading and trailing space characters are required because the answer is validated by a test of the form:

```
if  index(avalid_args(1), " " || answer || " ") > 0  then
     return ("1"b);
```

4. query_$af_validation
validates the answer by evaluating a command language active string. The active string is the only element of the avalid_args array. The answer is substituted into the active string, under control of the do active function. For example, the active string

```
                    [valid_pathname &f1 -min 1 -max 1 -exists segment]
```

would be evaluated as if

```
               [do "[valid_pathname &f1 -min 1 -max 1 -exists segment]" answer]
```

had been typed in a command line. The active string must evaluate to "true" or "false", otherwise query_$af returns a nonzero code argument. If it evaluates to "true", then the answer is considered valid. Otherwise, it is considered invalid. In the active string above, the answer would be considered valid if it contains one and only one valid pathname identifying an existing segment.

Entry: query_$ask

This entry point asks the user one of the questions defined by a previous call to the query_ entry point. It returns the user's answer.

Usage

```
     declare query_$ask entry (ptr, fixed bin, (*) char(*) varying, ptr, ptr,
          fixed bin(21), fixed bin(35));

     call query_$ask (Pq, qid, info_prompt, Pask_ctl, Panswer, Lanswer, code);
```

where:

1. Pq (Input)
points to the query data base.

2. qid (Input)
is the identifier of the question to be asked.

3. info_prompt (Input)
is an array of character strings, any one of which the user can type on a line by itself to cause the information string associated with the question to be typed. query_$ask will then ask the user the question again. A single null string argument may be given to disable prompting.

4. Pask_ctl (Input)
points to the ask_ctl structure described under "Notes" below. This structure contains information used by query_$ask.

5. Panswer (Output)
is a pointer to the answer returned for the question. The answer is stored in the query data base. When the answer is no longer needed, the space it occupies can be freed by calling query_$free_answer.

6. Lanswer (Output)
is the length (in characters) of the answer to the question. If Lanswer is 0, the user did not .answer the question and the question's validation routine accepted this fact. However, space was allocated to hold the null string answer. This space can be freed by calling query_$free_answer when no longer needed.

7. code (Output)
is a standard status code indicating failure in questioning the user. The code may have any value: returned by iox_$put_chars when asking a question; or by iox_$get_line when reading an answer; or by the validation routine; or it may have the following values.

error_table_$unimplemented_version
the ask_ctl structure pointed to by Pask_ctl is not a supported version of the structure. The caller must set ask_ctl.version to Vask_ctl_1 before calling query_$ask. See "Notes" below for more information.

error_table_$noentry
the question identified by qid has not yet been defined.

Notes

The Pask_ctl pointer argument of query_$ask points to the structure shown below. This structure is declared in query_ask_ctl_.incl.pl1.

```
dcl  1 ask_ctl                        aligned based(Pask_ctl),
       2 version                      fixed bin,
       2 S,
        (3 brief,
         3 adelims)                   bit(1) unal,
         3 pad1                       bit(34) unal,
       2 Pask_iocb                    ptr,
       2 Panswer_iocb                 ptr,
     Pask_ctl                         ptr,
     Vask_ctl_1                       fixed bin int static
                                           options(constant) init (1);
```

where:

1. version
is the version number of this structure. It is currently 1. See the description of Vask_ctl_1 below.

2. S.brief
when set to "1"b indicates that the brief version of the question is to be asked, rather than the long version.

3. S.adelims
when set to "1"b, indicates that answer delimiters are to be printed following the question when it is asked.

4. pad1
   is reserved for future use. The caller must set this to ""b.

5. Pask_iocb
   points to the I/O Switch Control Block (IOCB) through which the question is is asked. It must be opened for stream_output.

6. Panswer_iocb
   points to the I/O Switch Control Block (IOCB) through which the answer is read. It must be opened for stream_input.

7. Pask_ctl
   points to the ask_ctl structure.

8. Vask_ctl_1
   is a named constant which should be used to check for a structure version number of 1.

Entry: query_$add_unit

This entry point groups a series of questions together into a unit. Then query_$parse_unit can be called to parse an input segment looking for answers to all questions in the unit. Similarly, query_$ask_unit can be called to ask the user all of the questions in the unit.

When each unit is defined, space for a structure pointing to all of its answers is allocated in the query data base. A pointer to this structure is returned to the caller to identify the unit in subsequent calls to query_$parse_unit and query_$ask_unit. The structure is declared in query_unit_.incl.pl1 as follows.

```
dcl  1 query_unit        aligned based(Pquery_unit),
       2 version         fixed bin,
       2 Nanswers        fixed bin,
       2 answers (Nquery_unit_answers refer (query_unit.Nanswers)),
         3 P             ptr,
         3 L             fixed bin(21),
         3 qid           fixed bin,
         3 line_no       fixed bin,
         3 code          fixed bin(35),
         3 pad1 (2)      fixed bin,
     (QUESTION_ANSWERED  init(0),
      QUESTION_PREANSWERED
                         init(1),
      QUESTION_NOT_ANSWERED
                         init(2),
      QUESTION_ANSWERED_INCORRECTLY
                         init(3)) fixed bin internal static
                              options(constant),
      Pquery_unit        ptr,
      Vquery_unit_1      fixed bin internal static options(constant)
                              initial(1),.
      Nquery_unit_answers fixed bin;
```

where:

1. version

   is the version number of this structure. It is current 1. The variable Vquery_unit_1 should be used to check this version number (see 15 below).

2. Nanswers

   gives the number of questions/answers grouped together in the unit, and therefore determines the size of the unit structure.

3. answers

   is an array of minor structures, each element of which defines an answer for one of the questions. The answers are given in the order in which the questions were defined in the query_unit. The user may pre-answer questions to avoid asking a question in the group while still allowing the pre-answer to be included in the output generated by query_$format_unit.

4. p

   points to the answer for a question. When pre-answering a question, this should point to the first letter of the caller's answer.

5. l

   is the length (in characters) of the answer. When pre-answering a question, this should equal the length of the caller's answer.

6. qid

   is the identifier of the question associated with this answer. This is set by query_$add_unit and should not be changed by the caller.

7. line_no

   is the line number of the line on which query_$parse_unit found the beginning of the question/answer pair.

8. code

   is a code indicating whether the question has been answered. It may have to one of the values: QUESTION_ANSWERED, QUESTION_PREANSWERED, QUESTION_NOT_ANSWERED, QUESTION_ANSWERED_INCORRECTLY (see 10, 11, 12, 13 below). query_$add_unit sets code to QUESTION_NOT_ANSWERED. query_$ask_unit sets code to QUESTION_ANSWERED; but when an error occurs while asking the question, is sets code to the standard status code value returned by query_$ask. query_$free_answer and query_$free_unit_answers set code to QUESTION_NOT_ANSWERED when an answer is freed; but when the storage occupied by the answer is not found in the query data base, they set code to error_table_$not_done. When pre-answering a question, the caller should set code to QUESTION_PREANSWERED.

9. pad1

   is a reserved field.

10. QUESTION_ANSWERED

    is a named constant that can be compared with code to see if the question was answered correctly by a call to query_$ask_unit or query_$parse_unit.

11. QUESTION_PREANSWERED

    is a named constant that can be used to set code when the caller pre-answers a question.

12. QUESTION_NOT_ANSWERED

    is a named constant that can be compared with code to see if the question has not yet been answered.

13. QUESTION_ANSWERED_INCORRECTLY
is a named constant that can be compared with code to see if the question was answered incorrectly by a call to query_$parse_unit. The answer is returned, even though incorrect.

14. Pquery_unit
points to the query_unit structure. .

15. Vquery_unit_1
is a named constant that can be compared with version to insure that a version 1 structure is returned by query_$add_unit.

16. Nquery_unit_answers
is used to set the number of questions which are answered in the unit when the query_unit structure is allocated by query_$add_unit.

When a question has been pre-answered or answered by calling query_$ask_unit or query_$parse_unit, then that question will not be asked in subsequent calls to query_$ask_unit until a pre-answered question is marked QUESTION_NOT_ANSWERED or until the answer of a previously-asked question is freed by calling query_$free_answer, or query_$free_unit_answers. Similarly, query_$parse_unit will not look for the answer to such a question when it is parsing an input segment.

Usage

```
declare query_$add_unit entry (ptr, char(*), ptr, fixed bin(35));

call query_$add_unit (Pq, query_group, Pquery_unit, code);
```

where:

1. Pq (Input)
points to the query data base.

2. query_group (Input)
is a character string which identifies the questions to be grouped together in the unit. It contains a list of question identifiers, or question identifier ranges, separated by spaces. A question identifier is just an integer. A range of question identifiers is a pair of integers separated by a colon. For example, the query_group

"1 3 5:9 3 13:11 15"

groups together questions 1, 3, 5, 6, 7, 8, 9, 3, 13, 12, 11, and 15 into a unit in that order.

3. Pquery_unit (Output)
points to the query_unit structure for the new unit defined in this call.

4. code (Output)
is a standard status code which indicates the failure of unit definition. It may have one of the following values.

error_table_$bad_arg
the query_group does not define any questions.

error_table_$noentry
One or more of the questions identified in the query_group has not been defined in a call to the query_ entry point.

error_table_$bad_conversion
A syntax error or nonnumeric question identifier was found in the query_group.

Entry: query_$parse_unit

This entry point parses an input segment, looking for answers to all of the questions in a unit. Answers appear in the input segment, preceded by their question as an identifier. For example, the question "Date" with question delimiter of ":" and answer delimiter of ";" might appear in the input segment as

```
     Date:  November 17, 1978 ;
```

Either the long or short version of the question may identify an answer. Any of the question and answer delimiters may delimit the question and answer. Note that whitespace characters (space, horizontal-tab, vertical-tab, newline, newpage) appearing after the question delimiter are trimmed off the answer. The same is true for whitespace characters preceding the answer delimiter.

As the input segment is parsed, the answers found for questions are copied into the query data base to preserve their value, even if the input segment is modified. The values of the query_unit.answer minor structure are set to identify the answer. In particular, query_unit.answer.code is set to QUESTION_ANSWERED or QUESTION_ANSWERED_INCORRECTLY for answers found during the parse.

When parsing the input, questions appearing more than once in the unit are answered in their order of appearance in the unit. Answers for questions not appearing in the unit are ignored if the S.allow_unknowns flag is set. Otherwise, they are reported as errors to the user. Similarly, duplicate answers for the same question are ignored if the S.allow_duplicates flag is set. Otherwise, they are reported to the user as errors.

query_$parse_unit answers only those questions which have not been previously answered (i.e., it answers questions whose query_unit.answer.code is QUESTION_NOT_ANSWERED). Answers appearing in the input segment for previously answered questions are considered to be duplicates. To reparse previously answered questions, call query_$free_answer, or query_$free_unit_answers to free answers supplied by query_$parse_unit or query_$ask_unit. Set query_unit.answer.code to QUESTION_NOT_ANSWERED for pre-answered questions (those with a code of QUESTION_PREANSWERED).

When the answers are no longer needed, call query_$free_unit_answers to free the storage which the answers occupy in the query data base.

Usage

```
declare query_$parse_unit entry (ptr, ptr, ptr, ptr, fixed bin(21),
     fixed bin(35));

call query_$parse_unit (Pq, Pquery_unit, Pparse_unit_ctl, Pinput, Linput,
     code);
```

where:

1. Pq (Input)
   points to the query data base.

2. Pquery_unit (Input)
   points to the unit whose questions are to be answered by parsing.

3. Pinput (Input)
   points to the input segment to be parsed.

4. Linput (Input)
   is the length (in characters) of the input segment to be parsed.

5. Pparse_unit_ctl (Input)
   points to the parse_unit_ctl structure described under "Notes" below. This structure contains information used by query_$parse_unit.

6. code (Output)
   is a standard status code describing the failure of the parsing. It may have any value returned by an answer validation routine, or one of the following values.

   error_table_$unimplemented_version
      the parse_unit_ctl structure pointed to by Pparse_unit_ctl is not a supported version of the structure. The caller must set parse_unit_ctl.version to Vparse_unit_ctl_1 before calling query_$parse_unit. See "Notes" below for more information.

   error_table_$zero_length_seg
      a value of 0 was passed for Linput.

   query_et_$data_missing
      the input segment does not contain any non-whitespace characters.

   query_et_$data_duplicated
      duplicate answers were found for some questions in the query_unit, and parse_unit_ctl.S.duplicate_answers was "0"b. The error was reported to the user in an error message.

   error_table_$data_improperly_terminated
      answers for some questions in the query_unit were not terminated with the correct answer delimiter. The remainder of the input segment was used as the answer, and the error was reported to the user in an error message.

   query_et_$data_invalid
      answers for some questions in the query_unit were invalid. The invalid answer is returned, but query_unit.answer.code is set to QUESTION_ANSWERED_INCORRECTLY for such answers. The error was reported to the user in an error message.

query_et_$data_unknown

an unknown question was found in the input segment. An attempt was made to find the next known question and to continue parsing the input segment. The error was reported to the user in an error message if parse_unit_ctl.S.allow_unknowns was "0"b.

Notes

The Pparse_unit_ctl pointer argument pf query_$parse_unit points to the structure shown below. This structure is declared in query_parse_unit_ctl_.incl.pl1.

```
dcl  1 parse_unit_ctl              aligned based (Pparse_unit_ctl),
       2 version                   fixed bin,
       2 S,
        (3 allow_unknowns,
         3 duplicate_answers)      bit(1) unal,
         3 pad1                    bit(34) unal,
       2 Perror_iocb               ptr,
     Pparse_unit_ctl               ptr,
     Vparse_unit_ctl_1             fixed bin int static
                                   options(constant) init (1);
```

where:

1. version

   is the version number of this structure. It is currently 1. See the description of Vparse_unit_ctl_1 below.

2. S.allow_unknowns

   when set to "1"b, causes unknown answers (answers whose questions are not defined in the unit) to be ignored. Normally, such unknown answers are reported to the user as errors.

3. S.allow_duplicates

   when set to "1"b, causes duplicate answers to be ignored. Duplicate answers are those whose questions appear more times in the input segment than in the unit, or are questions which have been previously answered but which also appear in the input segment. Normally, duplicate answers are reported to the user as an error.

4. pad1

   is reserved for future use. The caller must set this to ""b.

5. Perror_iocb

   points to the I/O Switch Control Block (IOCB) through which an error can be reported to the user. The IOCB must be opened for stream_output.

6. Pparse_unit_ctl

   points to the parse_unit_ctl structure.

7. Vparse_unit_ctl_1

   is a named constant which should be used to check for a structure version number of 1.

Entry: query_$ask_unit

This entry point asks the user questions in a unit. The answers are copied into the query data base, and the values of the query_unit.answer minor structure are set to identify each answer. In particular, query_unit.answer.code is set to QUESTION_ANSWERED for each question which is answered.

Each time query_$ask_unit is called, the user is asked all unanswered questions in the unit. Unanswered questions are those whose query_unit.answer.code value is QUESTION_NOT_ANSWERED. In asking the question, this entry point types the brief or long version of the question (depending upon the setting of ask_unit_ctl.S.brief), then it types the first question delimiter defined for the question. The user then types his answer, followed by any one of the answer delimiters defined for the question. The answer is passed to the question's validation routine. If invalid, the information string describing the question is typed, then the user is asked the question again.

query_$ask_unit calls query_$ask to ask each question in the query_unit. When a question is answered, query_unit.answer.code is set to QUESTION_ANSWERED for that question, unless query_$ask returned an nonzero status code for that question. In that case, query_unit.answer.code is set to that status code.

query_$ask_unit asks only those questions which have not been previously answered (i.e., questions with a value of query_unit.answer.code of QUESTION_NOT_ANSWERED). To ask previously answered questions again, use query_$free_answer, or query_$free_unit_answers to release the storage occupied in the query data base by answers supplied by query_$parse_unit or query_$ask_unit. Set query_unit.answer.code to QUESTION_NOT_ANSWERED for pre-answered questions (those with a code of QUESTION_PREANSWERED). When the answers are no longer needed, call query_$free_unit_answers to free the storage which the answers occupy in the query data base.

Usage

```
     declare query_$ask_unit entry (ptr, ptr, (*) char(*) varying, ptr,
          fixed bin(35));

     call query_$ask_unit (Pq, Pquery_unit, Pask_iocb, Panswer_iocb,
          info_prompt, Pask_unit_ctl, code);
```

where:

1. Pq (Input)
   points to the query data base.

2. Pquery_unit (Input)
   points to the unit whose questions are to be asked.

3. info_prompt (Input)
   is an array of character strings, any one of which the user may type to ask to be prompted with the information string describing the question. After prompting, the question is repeated. A single null string argument may be given to disable the prompting.

4. Pask_unit_ctl (Input)
   points to the ask_unit_ctl structure described under "Notes" below. This structure contains information used by query_Sask_unit.

5. code (Output)
   is a standard status code describing the failure of question asking. It may have any value returned by query_Sask, or it may have the following value.

   error_table_Sunimplemented_version
   the ask_unit_ctl structure pointed to by Pask_unit_ctl is not a supported version of the structure. The caller must set ask_unit_ctl.version to Vask_unit_ctl_1 before calling query_Sask_unit. See "Notes" below for more information.

Notes

The Pask_unit_ctl pointer argument of query_Sask_unit points to the structure shown below. This structure is declared in query_ask_unit_ctl_.incl.pl1.

```
dcl  1 ask_unit_ctl                     aligned based(Pask_unit_ctl),
       2 version                        fixed bin,
       2 S,
        (3 brief,
         3 adelims)                     bit(1) unal,
         3 pad1                         bit(34) unal,
       2 Pask_iocb                      ptr,
       2 Panswer_iiocb                  ptr,
     Pask_unit_ctl                      ptr,
     Vask_unit_ctl_1                    fixed bin int static
                                             options(constant) init (1);
```

where:

1. version
   is the version number of this structure. It is currently 1. See the description of Vask_unit_ctl_1 below.

2. S.brief
   when set to "1"b indicates that the brief version of the question is to be asked, rather than the long version.

3. S.adelims
   when set to "1"b, indicates that answer delimiters are to be printed following the question when it is asked.

4. pad1
   is reserved for future use. The caller must set this to ""b.

5. Pask_iocb
   points to an I/O Switch Control Block (IOCB) through which questions are asked. The switch must be opened for stream_output.

6. Panswer_iocb
points to an I/O Switch Control Block (IOCB) through which the user's answers are read. The switch must be opened for stream_input.

7. Pask_unit_ctl
points to the ask_unit_ctl structure.

8. Vask_unit_ctl_1
is a named constant which should be used to check for a structure version number of 1.

<u>Entry</u>: query_$free_answer

This entry point frees the storage used for an answer obtained by calling query_$ask, query_$ask_unit or query_$parse_unit.

<u>Usage</u>

```
declare query_$free_answer entry (ptr, ptr, fixed bin, ptr, fixed bin(21),
     fixed bin(35));

call query_$free_answer (Pq, Pquery_unit, qid, Panswer, Lanswer, code);
```

where:

1. Pq (Input)
points to the query data base.

2. Pquery_unit (Input)
points to the query_unit structure for the unit containing the answer to be freed, when the question was answered by query_$ask_unit or query_$parse_unit. A null pointer should be given when freeing an answer obtained from query_$ask.

3. qid (Input)
is the identifier of the question which was asked.

4. Panswer (Input)
points to the storage for the answer to be freed.

5. Lanswer (Input)
is the length (in characters) of the answer to be freed.

6. code (Output)
is a standard status code indicating the failure of the freeing. It may have any of the following values.

error_table_$noentry
the question defined by qid has not been defined by a call to the query_ entry point or does not appear in the query_unit.

error_table_$not_done
no storage was found in the query data base for the answer to the question.

Entry: query_$free_unit_answers

This entry point releases query data base storage occupied by the answers in a unit. Only unit answers supplied by query_$ask_unit or query_$parse_unit occupy storage. query_unit.answer.code is set to QUESTION_ANSWERED or QUESTION_ANSWERED_INCORRECTLY for these questions. Pre-answered questions in the unit (those with query_unit.answer.code = QUESTION_PREANSWERED) are not changed.

Usage

```
     declare query_$free_unit_answers entry (ptr, ptr, fixed bin(35));

     call query_$free_unit_answers (Pq, Pquery_unit, code);
```

where:

1. Pq (Input)
   points to the query data base.

2. Pquery_unit (Input)
   points to the unit whose questions are to be freed.

3. code (Output)
   is a standard status code describing the failure of the freeing. It may have any value returned by query_$free_answer.

Entry: query_$format_unit

This entry point writes questions and answers associated with a unit into a segment in a format which can subsequently be parsed by query_$parse_unit. The questions are added to the segment in the order in which they were grouped in the unit by the query_$add_unit call.

For each question with a value of query_unit.answer.code of QUESTION_ANSWERED or QUESTION_PREANSWERED, the long version of the question is added to the segment (unless the $brief control argument is "1"b), followed by the first question delimiter, the answer, and the first answer delimiter. Unanswered questions are not put in the segment. Incorrectly answered questions are put in the segment only when format_unit_ctl.S.incorrect_answers is "1"b.

Usage

```
     declare query_$format_unit entry (ptr, ptr, ptr, ptr, fixed bin(21),
          fixed bin(21), fixed bin(35));

     call query_$format_unit (Pq, Pquery_unit, Pformat_unit_ctl, Pseg, Lin,
          Lout, code);
```

where:

1. Pq (Input)
   points to the query data base.

2. Pquery_unit (Input)
   points to the unit which is to be formatted.

3. Pformat_unit_ctl (Input)
   points to the format_unit_ctl structure described under "Notes" below. This structure contains information used by query_$format_unit.

4. Pseg (Input)
   is a pointer to the segment in which the formatted unit is to be placed. The unit can be appended to the end of existing data by setting the Lin argument, as described below. If Pseg is a null pointer, get_temp_segment_ is called to obtain a temporary segment in which the formatted unit is placed. The caller is then responsible for calling release_temp_segment_ to release this segment.

5. Lin (Input)
   is the length (in characters) of data already existing in the segment. The formatted unit is appended after this data. A value of 0 should be given to overwrite the segment. This value is assumed to be 0 if Pseg = null.

6. Lout (Output)
   is the length (in characters) of the segment after the formatted unit has been appended.

7. code (Output)
   is a standard status code describing the failure of unit formatting. It may have any value returned by get_temp_segment, or one of the following values.

   error_table_$unimplemented_version
      the format_unit_ctl structure pointed to by Pformat_unit_ctl is not a supported version of the structure. The caller must set format_unit_ctl.version to Vformat_unit_ctl_1 before calling query_$format_unit. See "Notes" below for more information.

   error_table_$out_of_bounds
      the segment in which the formatted unit was placed has overflowed. Lout is set to indicate how much data is returned, but some data may be lost. In particular, the final question/answer pair which was output may be incomplete.

Notes

The Pformat_unit_ctl pointer argument of query_$format_unit points to the structure shown below. This structure is declared in query_format_unit_ctl_.incl.pl1..

```
     dcl  1 format_unit_ctl            aligned based(Pformat_unit_ctl),
            2 version                  fixed bin,
            2 S,
             (3 brief,
              3 incorrect_answers)     bit(1) unal,
          Pformat_unit_ctl             ptr,
          Vformat_unit_ctl_1           fixed bin int static
                                             options(constant) init (1);
```

where:

1. version
   is the version number of this structure. It is currently 1. See the description of Vformat_unit_ctl_1 below.

2. S.brief
   when set to "1"b indicates that the brief version of the question is to be used in the formatted output, rather than the long version.

3. S.incorrect_answers
   when set to "1"b indicates that incorrectly answered question/answer pairs are to be placed in the formatted output, in addition to correctly answered pairs.

4. Pformat_unit_ctl
   points to the format_unit_ctl structure.

5. Vformat_unit_ctl_1
   is a named constant which should be used to check for a structure version number of 1.

Entry: query_$term

This entry point is called to terminate the query data base when all questioning is complete.

Usage

```
     declare query_$term entry (ptr);

     call query_$term (Pq);
```

where Pq points to the query data base.

Example

The following program excerpt illustrates the use of several query_ entry points.

```
census: proc;                  /* procedure to prompt for census data. */
          .
          .
          .
     dcl (Lanswer, Ltemp)      fixed bin(21),
         (Panswer, Pcensus_unit, Pq, Ptemp)
                               ptr,
          bc                   fixed bin(24);
     dcl  answer               char(Lanswer) based(Panswer);
     dcl  DOT (1)              char(3) internal static options(constant)
                                         init("\012.\012"),
          HT_SP_NL (3)         char(1) internal static options(constant)
                                         init("\011", " ", "\012"),
          NL (1)               char(1) internal static options(constant)
                                         init("\012"),
          QM (1)               char(2) internal static options(constant)
                                         init("?\012");

%include query_ask_ctl_;
     dcl  1 my_ask_ctl         automatic like ask_ctl;
%include query_ask_unit_ctl_;
     dcl  1 my_ask_unit_ctl    automatic like ask_unit_ctl;
%include query_$format_unit_ctl_;
     dcl  1 my_format_unit_ctl automatic like format_unit_ctl;
%include query_parse_unit_ctl_;
     dcl  1 my_query_parse_unit_ctl
                               automatic like query_parse_unit_ctl;

          Pq = null;           /* be prepared to clean up if census     */
          Ptemp = null;        /* taking is aborted.                    */
          on cleanup begin;
               if Ptemp ^= null then
                    call release_temp_segment_ ("census", Ptemp, code);
               if Pq ^= null then call query_$term (Pq);
               end;

          call query_$init ("census", Pq, code);
          if code ^= 0 then ....
                               /* create query data base.               */

                               /* define 4 census questions.            */
          call query_ (Pq, 1, "Person's Name", "Name",
               "Enter name of person being surveyed by the census.",
               ":", NL, nothing, "", query_$any_value, "", code);
          if code ^= 0 then ....
          call query_ (Pq, 2, "Person's Address", "Address",
               "Enter street address, city, state, zip, PO Box or Apt No.",
               ":", DOT, nothing, "", query_$any_value, "", code);
          if code ^= 0 then ....
          call query_ (Pq, 3, "Person's Age", "Age",
               "Enter person's age in years", ":", HT_SP_NL, nothing, "",
               query_$af_validation,
               "[valid_number &f1 -min 1 -max 1 -integer -from 1 -to 150]",
               code);
          if code ^= 0 then ....
          call query_ (Pq, 4, "Person's Occupation", "Occupation",
               "Enter occupation from known occupation list.",
               ":", NL, nothing, "", census_$validate_occupation,
               ">udd>CENSUS>data>known_occupations", code);
          if code ^= 0 then ....
          call query_$add_unit (Pq, "1:4", Pcensus_unit, code);
          if code ^= 0 then ....
                              /* group questions 1 thru 4 into a unit */
                              /* so we can ask, format and parse all   */
                              /* at one time.                          */

          my_ask_unit_ctl.version = Vask_ctl_1;
          my_ask_unit_ctl.S = "0"b;
          my_ask_unit_ctl.S.adelims = "1"b;
          my_ask_unit_ctl.Pask_iocb = iox_$user_output;
          my_ask_unit_ctl.Panswer_iocb = iox_$user_input;
          call query_$ask_unit (Pq, Pcensus_unit, QM,
               addr(my_ask_unit_ctl), code);
                              /* ask census taker all four questions. */

          my_format_unit_ctl.version = Vformat_unit_ctl_1;
          my_format_unit_ctl.S = "0"b;
          my_format_unit_ctl.S.incorrect_answers = "1"b;
          call query_$format_unit (Pq, Pcensus_unit,
               addr(my_format_unit_ctl), Ptemp, Ltemp, code);
          call iox_$put_chars (iox_$user_output, Ptemp, Ltemp, code);
                              /* format/print answers to verify them. */
                              /* Since Ptemp is null, formatted output*/
                              /* is placed in a temp seg.              */

          call query_ (Pq, 5, "Edit the answers", "Edit",
               "Type ""yes"" or ""y"" to edit census data.
   Type ""no"" or ""n"" if data is correct.", "?", HT_SP_NL, nothing, "",
               query_$list_validation, " yes y no n ", code);
          if code ^= 0 then ....
                              /* prepare to ask if user wants          */
                              /* to edit the answers.                  */

          call hcs_$fs_get_path_name (Ptemp, dir, Ldir, ent, code);
          path = substr(dir,1,Ldir) || ">" || ent;
                              /* get pathname of temp seg to edit it. */

          my_ask_ctl.version = Vask_ctl_1;
          my_ask_ctl.S = "0"b;
          my_ask_ctl.S.adelims = "1"b;
          my_ask_ctl.Pask_iocb = iox_$user_output;
          my_ask_ctl.Panswer_iocb = iox_$user_input;
          call query_$ask (Pq, 5, QM, addr(my_ask_ctl),
               Panswer, Lanswer, code);
                              /* Ask if answers are to be edited?      */

          do while (substr(answer,1,1) = "y");
                              /* Loop until answers are satisfactory. */

               call query_$free_unit_answers (Pq, Pcensus_unit, code);
               if code ^= 0 then ....
                              /* free storage in query data base       */
                              /* occupied by current answers.          */

               call edm (path);
                              /* Use edm to edit the answers.          */

               call hcs_$status_mins (Ptemp, 0, bc, code);
               Ltemp = divide (bc, 9, 24, 0);
                              /* get length of edited answers.         */
```

```
     my_parse_unit_ctl.version = Vparse_unit_ctl_1;
     my_parse_unit_ctl.S = "0"b;
     my_parse_unit_ctl.Perror_iocb = iox_$user_output;
     call query_$parse_unit (Pq, Pcensus_unit,
          addr(my_parse_unit_ctl), Ptemp, Ltemp, code);
                    /* parse up the edited question/answer   */
                    /* pairs.  Make sure editing fixed       */
                    /* errors rather than creating them.     */

     if code = 0 then do;
          call query_$format_unit (Pq, Pcensus_unit,
               addr(my_format_unit_ctl), Ptemp, 0, Ltemp, code);
          call iox_$put_chars (iox_$user_output, Ptemp, Ltemp,
               code);
                    /* reformat and print edited answers.    */

          call query_$free_answer (Pq, null, 5, Panswer,
               Lanswer, code);
          if code ^= 0 then ....
          call query_$ask (Pq, 5, QM, addr(my_ask_ctl),
               Panswer, Lanswer, code);
          end;      /* ask census taker if data is ok now.   */
                    /* if query_$parse_unit found errors in  */
                    /* parsing, it reports the errors.  We   */
                    /* then re-edit without asking user.     */

     end;           /* once loop completes, both census      */
                    /* taker and query_$parse_unit are       */
                    /* happy with the answers.               */
.
.
.
end census;
```

:Info: valid_af: valid: 12/28/78 validating active functions

This info segment describes active functions which check a value to determin if it is a correctly formed object of a given type. These active functions include-

valid_date, vdt	valid_picture, vpic
valid_number, vnb	valid_word, vw
valid_pathname, vpn	

:Info: valid_word: vw: 12/28/78 valid_word, vw

Syntax: [vw {words} {-control_args}]

Function: validates a set of input words to insure that one or more of the words is found in a list of acceptable words, or in a named set of dictionaries. A value of true is returned if the words are valid; false is returned otherwise.

Arguments:

words
are zero, one or more words to be validated.

Control arguments:

-word STR
specifies that STR is a word, even though it looks like a control argument.

-all, -a
requires that all of the words are valid before a value of true is returned. A value of true is also returned if no words are given. (This is default.)

-any
requires that only one of the words is valid before a value of true is returned. A value of true is also returned if no words are given.

-maximum N, -max N
requires that no more than N words are given. If more than N are given, a value of false is returned whether or not the words are valid. (Default = infinite number of words.)

-minimum N, -min N
requires that at least N words are given. If fewer than N are given, a value of false is returned. (Default = 0)

-ignore_case
specifics that the case of letters is ignored when comparing the words with a list of acceptable answers or with dictionary entries. (Default, case matters).

-alphabetic, -aptha
requires that valid words consist of only letters of the alphabet.

-number, -nb
requires that valid words consist only of digits from 0 through 9.

-alphanumeric, -alphan
requires that valid words consist only of alphabetic letters or digits.

-identifier, -id
requires that valid words meet the constraints imposed upon identifiers in PL/I source programs.

-accept words
gives a list of acceptable words. At least one word must be given. All of the arguments following -accept are treated as part of the list. Thus -accept, if present, must be the last control argument.

-dictionary {paths}, -dict {paths}
gives pathnames of one or more dictionaries containing valid words. All arguments following -dict are treated as pathnames. Thus -dict, if present, must be the last control argument and is mutually exclusive with -accept. If no pathnames are given, the dictionaries given in the "dictionary" search list are used.

Notes: Control arguments in the following lines are mutually exclusive with other members of the line; only one member of each line may be used.
-any, -all
-alphabetic, -number, -alphanumeric, -identifier
-accept, -dictionary

Syntax as a command: vw {words} {-control_args}

Info: valid_pathname: vpn: 01/10/79 valid_pathname, vpn

Syntax: [vpn {paths} {-control_args}]

Function: validates a set of pathnames to insure that all pathnames are valid. Pathnames are valid if they are acceptable to the expand_pathname_ subroutine, and if they meet the existence criteria of the -exists control argument.

Arguments:
paths
are zero, one or more pathnames to be validated. The star convention is allowed in final entryname of path.

Control arguments:
-maximum N, -max N
requires that no more than N paths are given. If more than N are given, a value of false is returned whether or not the paths are valid. (Default = infinite number of paths.)

-minimum N, -min N
requires that at least N paths are given. If fewer than N are given, a value of false is returned. (Default = 0)

-exists type
checks to see if the pathnames exist in the storage system as a given type of entry. Any keyword given under "List of types" below may be given.

-chase
causes link targets to be checked for existence when -exist is given. -chase allowed only with -exists.

-all, -a

requires that all of the pathnames are valid and exist (when -exists is used) before a value of true is returned. A value of true is also returne if no pathnames are given. (This is default.)

-any

requires that only one of the pathnames is valid and exists before a value of true is returned. A value of true is also returned if no pathnames are given.

List of types:

branch

segment, multisegment file or directory must exist.

directory, dir

directory must exist.

entry

segment, multisegment file, directory or link must exist.

file

segment or multisegment file must exist.

link

link must exist.

master_directory, mdir

master directory must exist.

msf

multisegment file must exist.

nonbranch

link must exist.

nonfile

link or directory must exist.

nonlink

segment, directory or multisegment file must exist.

nonmaster_directory, nmdir

directory not a master directory must exist.

nonmsf

link, segment or directory must exist.

nonnull_link, nnlink

link must exist to an existing segment, directory or multisegment file.

nonsegment, nonseg

link, multisegment file or directory must exist.

nonzero_file, nzfile

segment or multisegment file must exist, must have nonzero bit count.

nonzero_msf, nzmsf

multisegment file must exist, must have nonzero bit count.

nonzero_segment, nzseg

segment must exist, must have nonzero bit count.

null_link

link must exist, link target must not exist.

segment, seg

segment must exist.

zero_file, zfile

segment or multisegment file must exist, must have zero bit count.

zero_msf, zmsf
multisegment file must exist, must have zero bit count.
zero_segment, zseg
segment must exist, must have zero bit count.

Notes: If any pathname is not accepted by expand_pathname_, then a value of false is returned.

The -any and -all control arguments are mutually exclusive; only one may be given.

Syntax as a command: vpn {paths} {-control_args}

:Info: valid_date: vdt: 01/10/79 valid_date, vdt

Syntax: [vdt {dates} {-control_args}]

Function: validates a set of date/time specifications to insure that all dates are valid and that one or more of the dates falls within a given time period. Date/time specifications are valid if they are acceptable to the convert_date_to_binary_ subroutine.

Arguments:
dates
are zero, one or more date/time specifications. If the specification includes spaces, it must be enclosed in quotes.

Control arguments:
-from date, -fm date
gives beginning of time period in which valid dates must fall. The time period includes the date/time specified by date (to the nearest microsecond). (Default - accept dates from January 1, 0000 00:00:00.000000 gmt)
-to date
gives end of time period in which valid dates must fall. The time period includes the date/time specified by date (to nearest microsecond). (Default - accept dates to December 31, 9999 23:59:59.999999 gmt)

-all, -a
requires that all of the dates fall within the given time period before a value of true is returned. A value of true is also returned if no dates are given. (This is default.)
-any
requires that only one of the dates falls within the given time period before a value of true is returned. A value of true is also returned if no dates are given.

-maximum N, -max N
requires that no more than N dates are given. If more than N are given, a value of false is returned whether or not the dates are valid. (Default = infinite number of dates.)

-minimum N, -min N
requires that at least N dates are given. If fewer than N are given, a value of false is returned. (Default = 0)

Notes: if any date is not acceptable to convert_date_to_binary_, then a value of false is returned.

Syntax as a command: vdt {dates} {-control_args}

:Info: valid_number: vnb: 01/10/79 valid_number, vnb

Syntax: [vnb {numbers} {-control_args}]

Function: validates character representations of numbers to insure that all are valid and that one or more numbers fall within a given range.

Arguments:

numbers
are zero, one or more character string representations of numbers. Integer, fixed-point or floating-point representations may be given. Numbers are assumed to be expressed in base 10, but may be expressed in base 2, 4, 8 or 16 by ending the representation with b, q, o or x respectively. For floating-point numbers, only the mantissa is expressed in a nondecimal base; the exponent must be expressed in decimal. This follows the PL/I convention for arithmetic constants.

Control arguments:

-range STR, -rg STR
defines a range in which valid numbers must fall. STR has one of the forms:

```
lower_bound<X<upper_bound
lower_bound<X
           X<upper_bound
```

where X is any alphabetic symbol representing the numbers being validated. lower_bound and upper_bound are numbers, as described above for number arguments. The relational operator <= may be used in place of < to specify inclusive ranges. If STR contains spaces, then it must be enclosed in quotes. A sample range is: ".314159265e+1 < X <= 99".
(Default: -infinity <= X <= infinity)

-fixed
requires that valid numbers be expressed as fixed-point character representations. A radix point and fractional digits are optional.

-integer
requires that valid numbers be expressed as integer character representations. A radix point and fractional digits are not allowed.

-float
requires that valid numbers be expressed as floating-point character representations. A radix point and fractional digits are optional, but an exponent is required.

-all, -a
requires that all of the numbers fall within the given range before a value of true is returned. A value of true is also returned if no numbers are given. (This is default.)

-any
requires that only one of the numbers falls within the given range before a value of true is returned. A value of true is also returned if no numbers are given.

-maximum N, -max N
requires that no more than N numbers are given. If more than N are given, a value of false is returned whether or not the numbers are valid. (Default = infinite number of numbers.)

-minimum N, -min N
requires that at least N numbers are given. If fewer than N are given, a value of false is returned. (Default = 0)

Notes: Control arguments in the following lines are mutually exclusive with other members of the line; only one member of each line may be used.
-fixed, -float, -integer
-all, -any

ntax as a command: vnb {numbers} {-control_args}

:Info: valid_pic: vpic: 01/10/79 valid_pic, vpic

Syntax: [vpic pic_spec {values} {-control_args}]

Function: checks to see if one or more values can be edited into a PL/I numeric or character pictured string. If no values are given, checks to see if given numeric or character pictured string is valid.

Arguments:
pic_spec
is a PL/I numeric or character pictured string (picture).
values
are one or more values to be edited into the picture. If no values are given, the pic_spec itself is checked for validity.

Control arguments:
-value STR, -vl STR
specifies that STR is a value to be edited into pic_spec, even though it looks like a control argument.

-all, -a
requires that all values can be correctly edited into pic_spec before a value of true is returned. (This is default.)

-any
requires that only one value can be correctly edited into pic_spec before a a value of true is returned.

-maximum N, -max N
requires that no more than N values are given. If more than N are given, a value of false is returned whether or not the values are valid. (Default = infinite number of values.)

-minimum N, -min N
requires that at least N values are given. If fewer than N are given, a value of false is returned. (Default = 0)

Notes: The -any and -all control arguments are mutually exclusive; only one may be given.

Syntax as a command: vpic pic_spec {values} {-control_args}

```
include   et_macros

et        query_et_

ec        data_duplicated,data_dup,
            (Duplicate data found.)
ec        data_invalid,data_inv,
            (Invalid data found.)
ec        data_missing,data_mis,
            (Expected data missing.)
ec        data_unknown,data_unk,
            (Unknown data values found.)

end
```